Model Checking classical mutual exclusion algorithms

Ivan Ivchenko

01/08/2024

Abstract

In computer science, mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions. It is the requirement that one thread of execution never enters a critical section while a concurrent thread of execution is already accessing said critical section, which refers to an interval of time during which a thread of execution accesses a shared resource or shared memory.

Mutual exclusion is crucial because it keeps shared resources safe and consistent. Without it, multiple threads might try to change the same data at the same time, causing errors and unpredictable results. By making sure only one thread can use the shared resource at a time, mutual exclusion helps maintain the correct operation of programs, especially those running multiple threads or processes simultaneously. This is vital for the reliability and stability of software systems.

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

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

Algorithm 2: Algorithm for process A

```
flag[0] := true
while flag[1] = true do
    if turn = 1 then
        flag[0] := false
        while turn = 1 do
            busy-wait
        end
        flag[0] := true
    end
end
// critical section
...
turn := 1
flag[0] := false
// end of critical section
...
```

Algorithm 3: Algorithm for process B

```
flag[1] := true
while flag[0] = true do
    if turn = 0 then
        flag[1] := false
        while turn = 0 do
            busy-wait
        end
        flag[1] := true
    end
end
// critical section
...
turn := 0
flag[1] := false
// end of critical section
...
```

Dekker's algorithm guarantees mutual exclusion, the impossibility of mutual locking or hang-up. Let's consider why the last property is true. Suppose that A has stayed inside the "while flag[1]" loop forever. Since mutual locking cannot occur, sooner or later p1 will reach its critical section and set turn = 0 (the value of turn will remain constant while B is not advancing). A will exit the "while turn = 1" inner loop (if it was there). It will then set flag[0] to true and wait for flag[1] to set to false (since turn = 0, it never performs an action in the "while" loop). The next time B tries to enter the critical section, it will be forced to execute actions in the "while flag[0]" loop. Specifically, it will set flag[1] to false and execute the "while turn = 0" loop (since turn remains 0). The next time control passes to A, it will exit the "while flag[1]" loop and enter the critical section.

If you modify the algorithm so that the actions in the "while flag[1]" loop are executed without checking the "turn = 0" condition, you will get the possibility of starvation, so all the steps of the algorithm are necessary.

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.

```
bool flag[2] = {false, false};
int turn = 0;
int critical_section = 0;
```

Figure 1: Global variables declarations

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

Figure 2: System declarations

Because we want to model a non-deterministic behavior of our algorithm, we should somehow indicate which turn it is without specifying it directly (Figure 1).

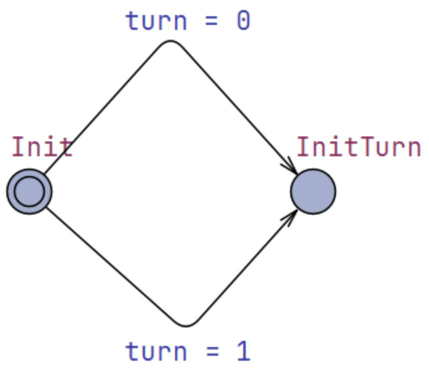


Figure 3: Initialization of turn var

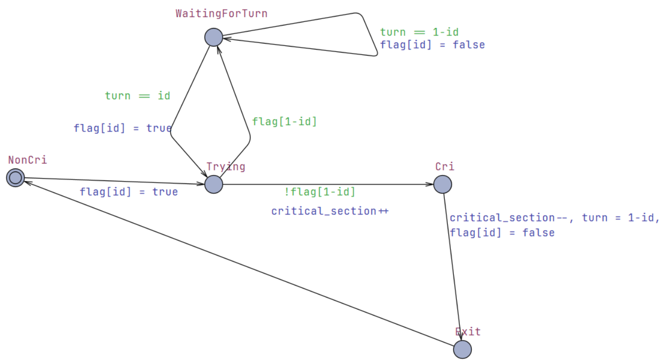


Figure 4: Definition of our algorithm

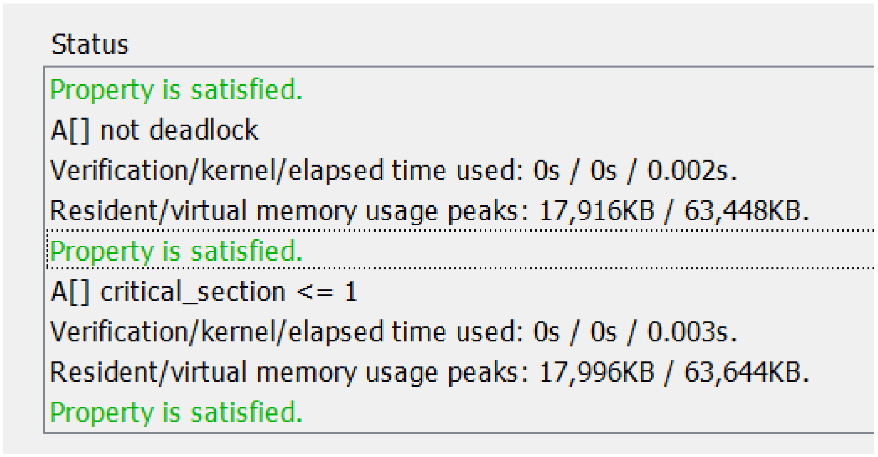
Now moving on to the process definition. For this I've simplified our model for a better understanding of the concept of algorithm, the guard statements (what should hold in order to move from one state to another) are marked green and the update statements (what values will be changed after the step) are marked blue (Figure 2).

For the properties to be verified we specify:

1. $A[]$ critical_section ≤ 1 - 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, if both processes would enter their critical sections at the same time this will result in incrementing our value by 2, which will trigger an error.

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

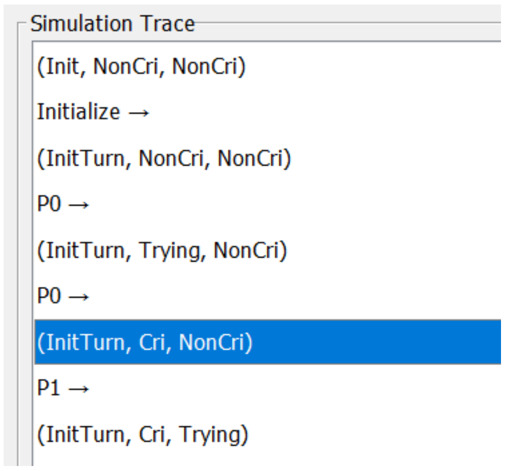
After this, all we have to do is to run a check to see if our properties are being satisfied.



```
Status
Property is satisfied.
A[] not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 17,916KB / 63,448KB.
Property is satisfied.
A[] critical_section <= 1
Verification/kernel/elapsed time used: 0s / 0s / 0.003s.
Resident/virtual memory usage peaks: 17,996KB / 63,644KB.
Property is satisfied.
```

Figure 5: Results of our check

Also, it is possible for us to see the traces of our run (it is presented as a tuple (S0,S1,S2) where S0 - state in 1st process (our initialization of turn) and S1 and S2 are Process1 and Process2 respectively). (Figure 6 and Figure 7)


Simulation Trace

(Init, NonCri, NonCri)

Initialize →

(InitTurn, NonCri, NonCri)

P0 →

(InitTurn, Trying, NonCri)

P0 →

(InitTurn, Cri, NonCri)

P1 →

(InitTurn, Cri, Trying)

Figure 6: Traces

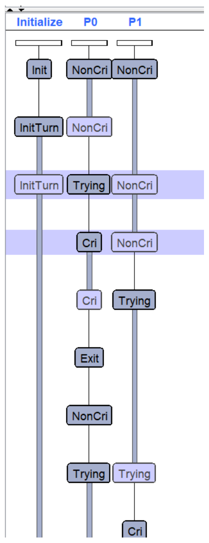


Figure 7: Traces (graphical)

0.3 Petterson's Algorithm

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 P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

The pseudocode for Petterson's and Dekker's algorithms are much the same, except for some details:

1. **turn** variable placement: In Dekker's algorithm the **turn** variable is checked and set within the loop checking the other process's flag, requiring a process to release and re-check its flag if the other process wants to enter the critical section. When in Peterson's Algorithm the **turn** variable is set before the busy-wait loop, allowing each process to check both the other process's flag and the turn variable in a single condition.

2. **flags** : In Dekker's algorithm the processes repeatedly set and unset their flags based on the turn variable, resulting in more complex flag manipulation while in Peterson's algorithm each process sets its flag once before entering the busy-wait loop and clears it after exiting the critical section, leading to more straightforward flag handling.

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

Algorithm 4: Initialization

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

Algorithm 5: Algorithm for process A

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

Algorithm 6: Algorithm for process B

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

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 Petterson's algorithm loaded into the tool and how i defined it.

```
bool flag[2] = {false, false};
int turn;
int critical_section = 0;
```

Figure 8: Global variables declarations

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

Figure 9: System declarations

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 10).

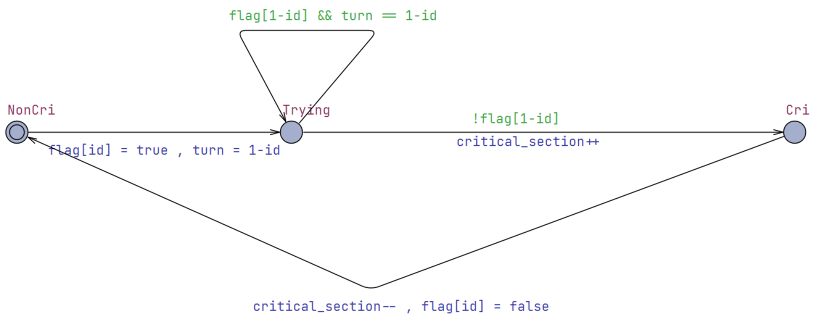
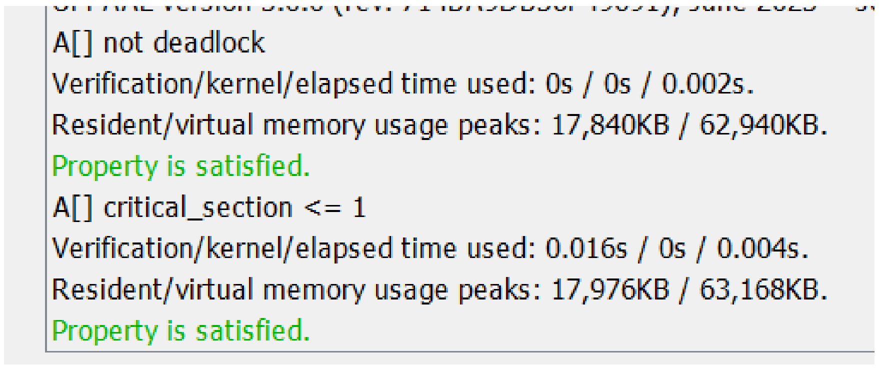

flag[1-id] && turn == 1-id
NonCri
Trying
!flag[1-id]
critical_section++
Cri
flag[id] = true , turn = 1-id
critical_section-- , flag[id] = false

Figure 10: Definition of our algorithm

For the properties to be verified we specify:

1. $A[]$ critical_section ≤ 1 - 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, if both processes would enter their critical sections at the same time this will result in incrementing our value by 2, which will trigger an error.
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

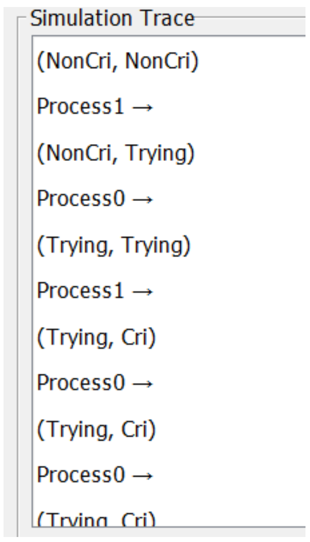
After this, all we have to do is to run a check to see if our properties are being satisfied.



```
A[] not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 17,840KB / 62,940KB.
Property is satisfied.
A[] critical_section <= 1
Verification/kernel/elapsed time used: 0.016s / 0s / 0.004s.
Resident/virtual memory usage peaks: 17,976KB / 63,168KB.
Property is satisfied.
```

Figure 11: Results of our check

Also, we can see the traces of our run (it is presented as a tuple (S0, S1) where S1 and S2 are Process1 and Process2 respectively). (Figure 12 and Figure 13)



```
Simulation Trace
(NonCri, NonCri)
Process1 →
(NonCri, Trying)
Process0 →
(Trying, Trying)
Process1 →
(Trying, Cri)
Process0 →
(Trying, Cri)
Process0 →
(Trying, Cri)
```

Figure 12: Traces

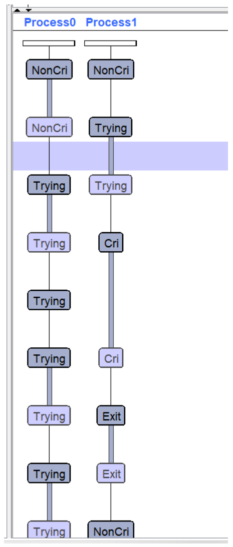


Figure 13: Traces (graphical)

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.