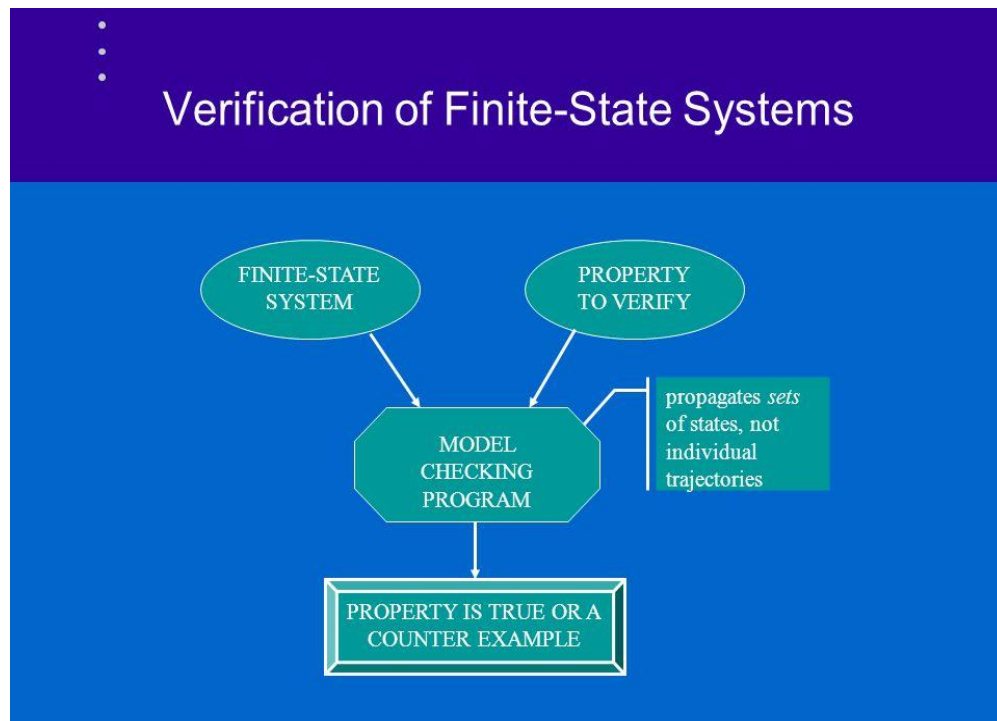


Electronic Design Automation
Lab 3
Verification using Model Checking
Name : Sudha Ravali Yellapantula
PID : 906109009

Overview :

Model checking is an automatic verification technique for finite state systems and is used for verifying the properties of the finite state systems. This was developed independently by Clarke and Emerson and by Queille and Sifakis in early 1980's.

In order to solve such a problem of finite state machine algorithmically, both the model of the system and the specification are formulated in some precise mathematical language. Specifications are written in propositional temporal logic. Verification procedure is an exhaustive search of the state space of the design. An important class of model checking methods have been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula.



The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state

machine (FSM), i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Verification Process

The things that make a software model checking tool different is that it can have large unbounded base types such as int, float, etc. They have user defined classes or types and procedure calls, recursion etc are observed. Moreover, concurrency and number of unbounded threads are seen in this, which most probably won't be witnessed in hardware systems. Software model checking involves the use of static analysis to extract a model K from a Boolean extraction program, followed by checking if a certain specification is true for a given model K .

The program is then simulated along the paths of the computation tree and then finite state machines are used to look for patterns in the control flow graph, softwares can be used to generate these state transition diagrams according to the code given. Since fixpoint computation is too expensive to compute a set of reachable states, the search is restricted to states that are reachable from initial state within fixed number n .

From the guidance given in the class, the software algorithm suited to model checking that I chose is that of Mutual Exclusion. The algorithm I proposed in the class is Peterson's algorithm and I stated that I would prove the correctness of the algorithm including its safety, liveness and correctness properties. After I was successful in doing so, I went on to create another, more complex version of mutual exclusion between two (can be extended to more threads as well) threads to try another method. After that I finally arrived at the Producer Consumer buffer problem whose correctness properties are all satisfied.

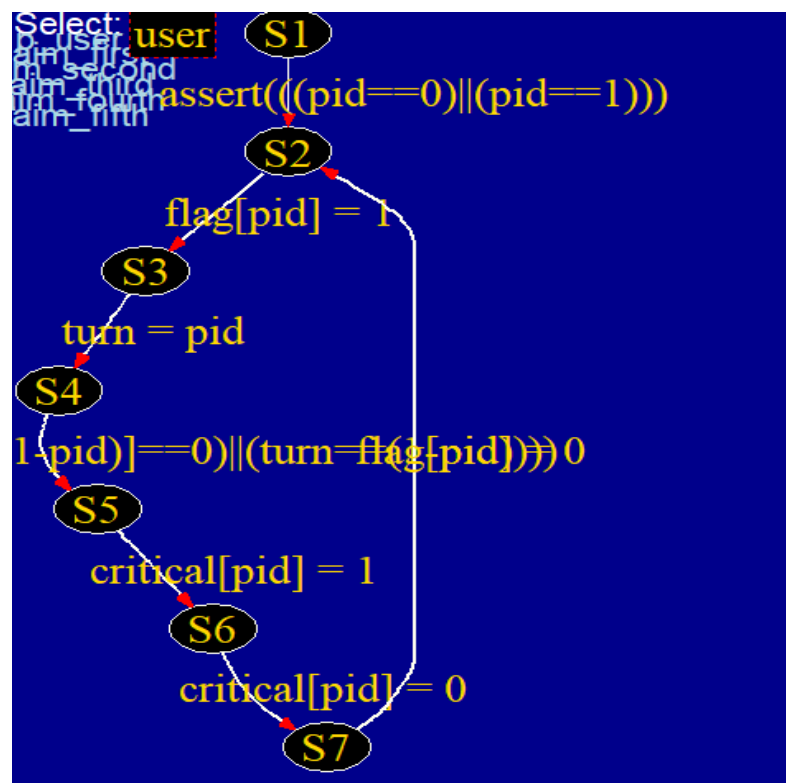
The model checking tool I used to help me with the execution of this project is SPIN – Model Checking tool. Spin takes in Promela language code as the input for the proper execution and verification of the algorithm and provides state diagrams for the given input. The input C code is first converted into promela by an application called modex. The state diagram thus obtained is useful in getting to know about the flow of the execution of the program. First the program is run to see if there are any errors in the program. In the basic Peterson's Mutual Exclusion problem, first I ran the given program to check if the conversion is done right and the program is getting executed without any errors. Later, I after careful observation and understanding of the program, I started adding the LTL properties to prove its safety, liveness and fairness properties.

During this time, it was always kept in mind not to violate any assertions made, to not get trapped by the deadlocks and care was taken to not run into errors while verifying the algorithm or its properties. Spin supports only LTL, so therefore all the specifications were mentioned using LTL properties. Any

errors encountered were always identified with the help of counter examples and using the trace, the bugs were always found and was helpful in identifying the faults with the properties or the code.

The LTL properties are like the claims made that proves the correctness of the algorithm mentioned. As long as all the properties are mentioned without any errors and are satisfying the correctness properties, the LTL properties are known to be working just fine and will prove to be useful in verifying the algorithm itself. SPIN provides an interface to stop and look at the automata given as well, which would give the user a better idea of the flow taken.

Automata obtained through SPIN for Peterson's Algorithm :



Here, we assert “ $\text{assert}(_pid == 0 \mid \mid _pid == 1)$ ”, ie, through every step of the way, our algorithm makes sure that this statement or expression is not violated throughout the program by our ltl properties. For this, we have no errors produced, all the invalid end states are checked and moreover the liveness and safety properties are checked as well. In this way, by proving these properties with the help of appropriate specifications, we formally verify the given algorithm.

Suitability of the Application for Model Checking

The application chosen is Mutual Exclusion, followed by implementation of the Producer Consumer buffer. The suitability of any application to be eligible for model checking is that the algorithm can be produced in terms of a state transition graph and that the system model and the system property should always be in relation to each other and must be satisfied. While discussing this, it must be kept in mind that State transition graphs are not necessarily finite-state. Also, finite state-transition graphs don't handle recursion or process creation.

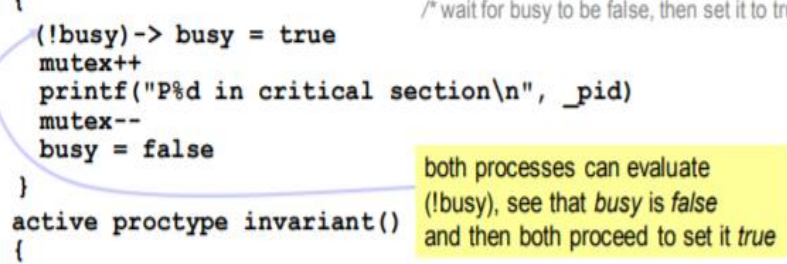
The application of "Mutual Exclusion" is one of the most popular methods for formal verification and model checking. This is because it enables us to check the working of a concurrent system where two or more threads try to access the variables in the critical section. A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

Safety property is that something "bad" will never happen, whereas Liveness property is that something "good" will happen at some point but we don't know when. Since it was possible to examine the correctness, safety and liveness properties for all the programs, while obtaining the suitable state diagrams for all the developmental algorithms as well and not pertaining to state explosion, the producer consumer buffer along with mutual exclusion is considered to be an extremely suitable for model checking.

For example,

```
bool busy          /* signal entering/leaving the section */
byte mutex        /* counts # procs in critical section */

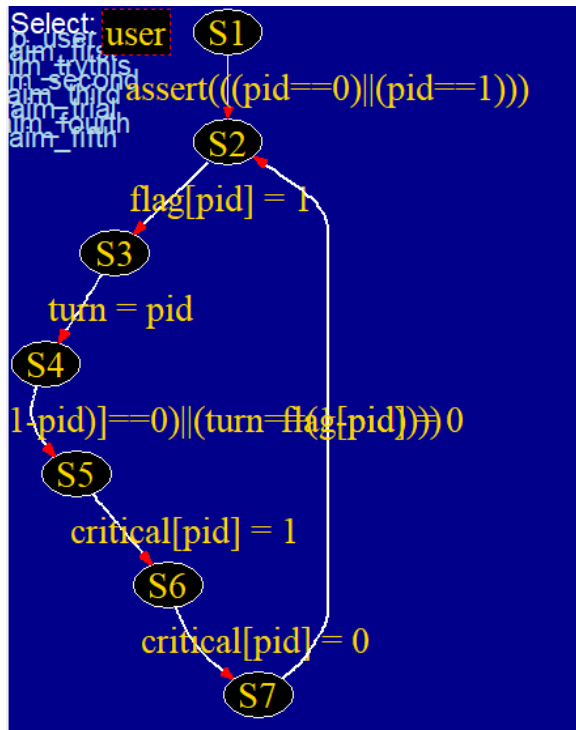
active [2] proctype P()
{
    /* wait for busy to be false, then set it to true */
    (!busy)-> busy = true
    mutex++
    printf("P%d in critical section\n", _pid)
    mutex--
    busy = false
}
active proctype invariant()
{
    assert(mutex <= 1)
}
```



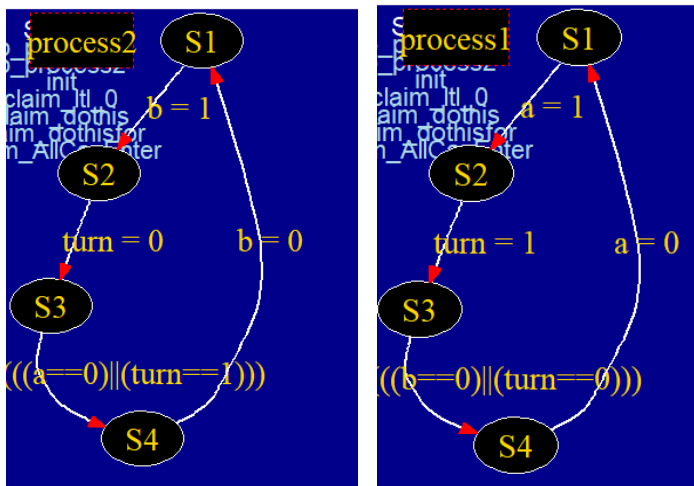
For all the different applications, the LTL properties state that the processes are never both in the critical section. Also, No matter what happens, a process will eventually get to a critical section. If process 0 is in the critical section, it means that process 1 would be the next.

All three if the applications could successfully be developed into state transition systems as below and everyone of them showcased the correctness property by maintaining the mutual exclusion property.

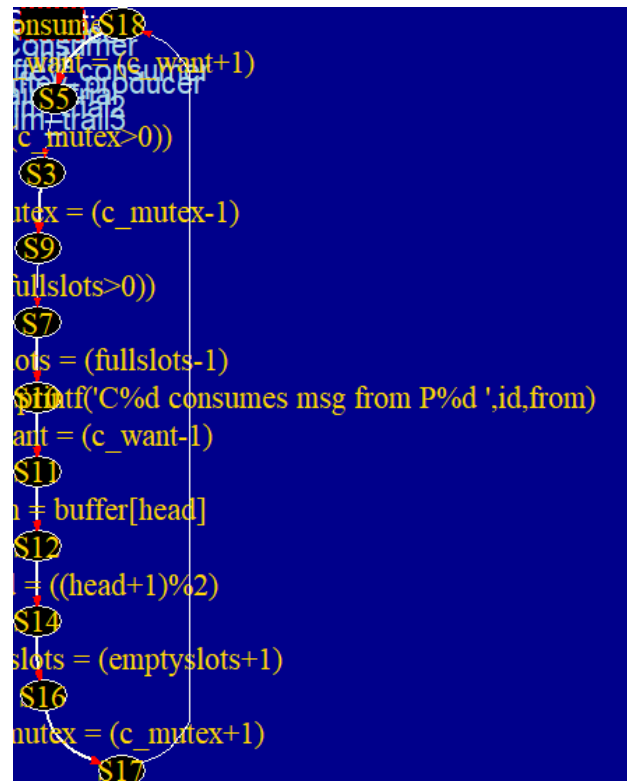
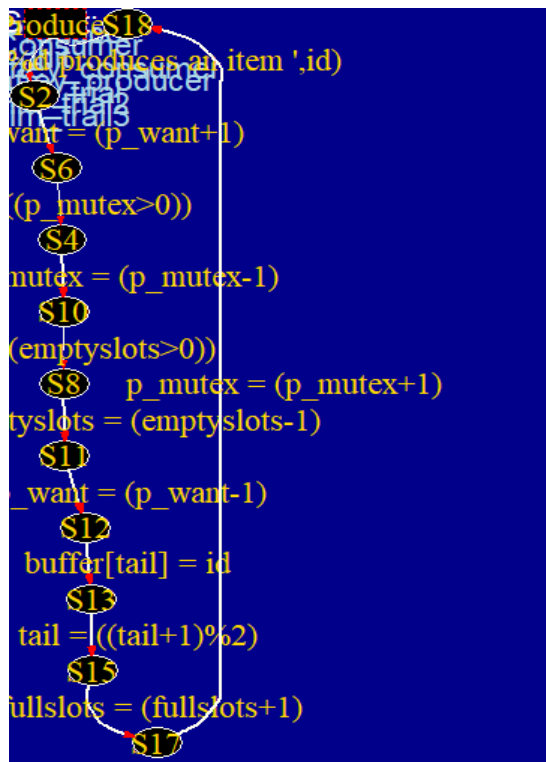
Peterson's Algorithm :



Application 2 : Developed implementation of Mutex :



Producer Consumer Buffer Problem :



Model Checking Tool Used

Spin is a popular open-source software verification tool, that can be used for formal verification of multi-threaded software applications. It has been developed at Bell Labs in the Unix group of Computing Sciences Research Center, starting in 1980 and has continued to evolve to keep pace with new developments. The Spin 6.4.8 version was used in the implementation of this algorithm.

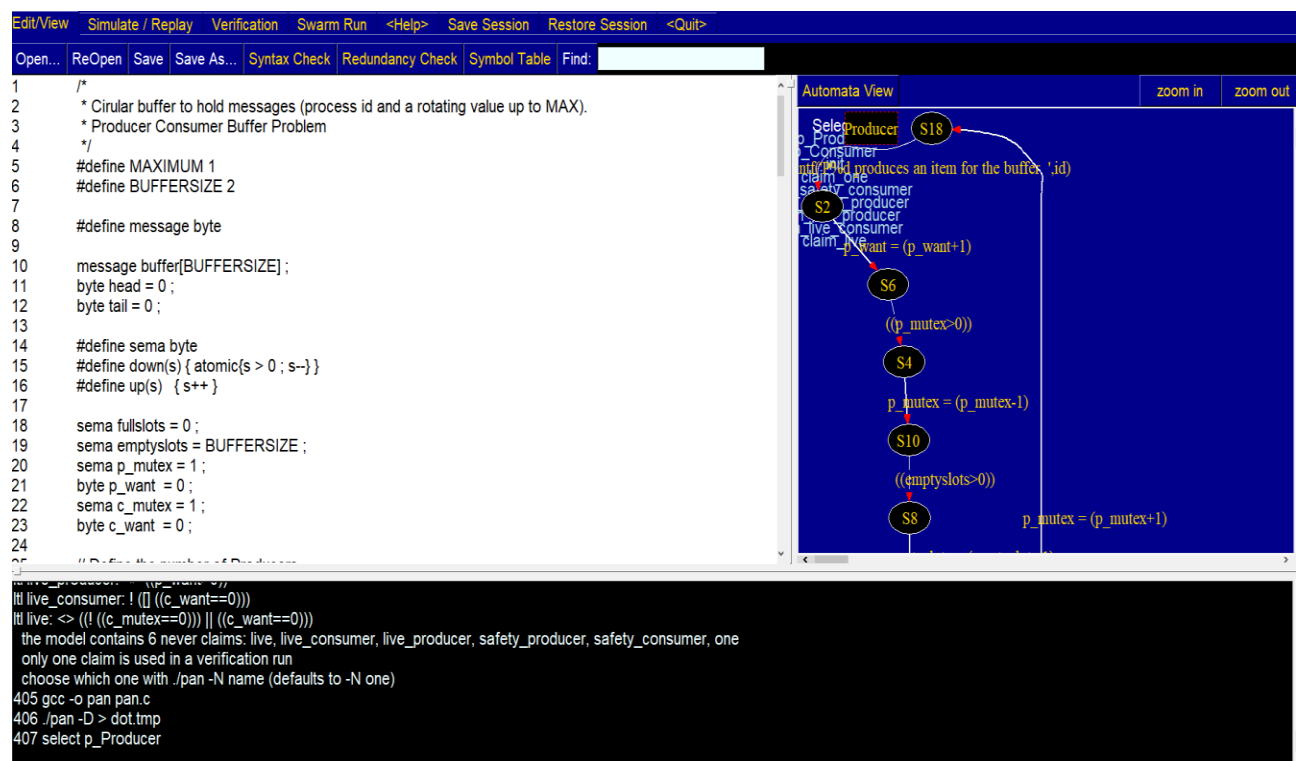
Spin targets the efficient verification of multi-threaded software, not the verification of hardware circuits. This tool supports a high level language, called Promela to specify systems Promela is short for Process Meta Language. Spin is also useful in tracing logical design errors in distributed systems, concurrent algorithms and data communications protocols.

Spin checks the logical consistency of a specification and reports on deadlocks, race conditions, different types of incompleteness etc in the verification section along with the time taken to

achieve them. Though model extractor is used to convert c code to promela, I had to make a few changes to the resultant code to make it syntactically and algorithmically error free.

The main reason for choosing this tool for Producer Consumer is that it supports model checking where concurrency is involved also provides us with valid state diagrams or automata as soon the code given as an input in promela. The promela files have an extension of (.pml). Spin provides direct support for the use of multi-core computers for model checking runs, thus supporting the verification of both safety and liveness properties.

Spin works on-the-fly, ie, it avoids the need to preconstruct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties. This makes it possible to verify very large system models. The global state graph obtained as soon as the input is given is shown below :



Spin is a full LTL model checking system, supporting all correctness requirements expressed in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. The Graphical User Interface for this is iSpin, and it has inbuilt options for checking the liveness, safety properties for the whole algorithm as well as for the claims separately.

Correctness properties in spin can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi automata, or more broadly as general

omega-regular properties in the syntax of Spin never claims. For this project, I have used both assertions, LTL properties and never claims as well to verify the correctness of the systems and it ran error free for all the cases.

The tool supports random, interactive and guided simulation, and both exhaustive and partial proof techniques, based on depth-first search, breadth-first search, or bounded context-switching. The tool is designed to scale well, and to handle large problem sizes efficiently. It can also be used as an exhaustive verifier, capable of rigorously proving the validity of user specified correctness requirements using partial order reduction theory to optimize the search

Instead of specifying a never claim, an LTL formula can be directly written inside a verification model. There is no restriction on how many properties can be specified. The formulae are converted by Spin into never claims in the background and the property that should be checked for any run can be chosen. Inline LTL properties state positive properties to prove, i.e., they do not need to be negated to find counter-examples, this may change from one version of Spin to another. All Spin software is written in ISO-standard C, and is portable across all versions of Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows.

Meaning of Specifications Used :

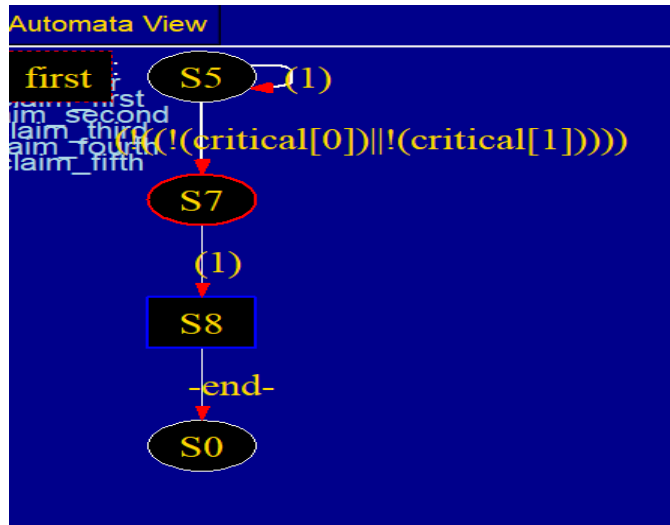
Spin makes use of the LTL, ie, Linear Temporal Logic to give its input specifications. A few notations that would be helpful in understanding the specifications that I have used for my project :

[]	(the temporal operator always)
<>	(the temporal operator eventually)
!	(the boolean operator for negation)
&&	(the boolean operator for logical and)
	(the boolean operator for logical or)
U	(the temporal operator strong until)

For the first basic program which is the Peterson's algorithm for mutual exclusion, the specifications used are :

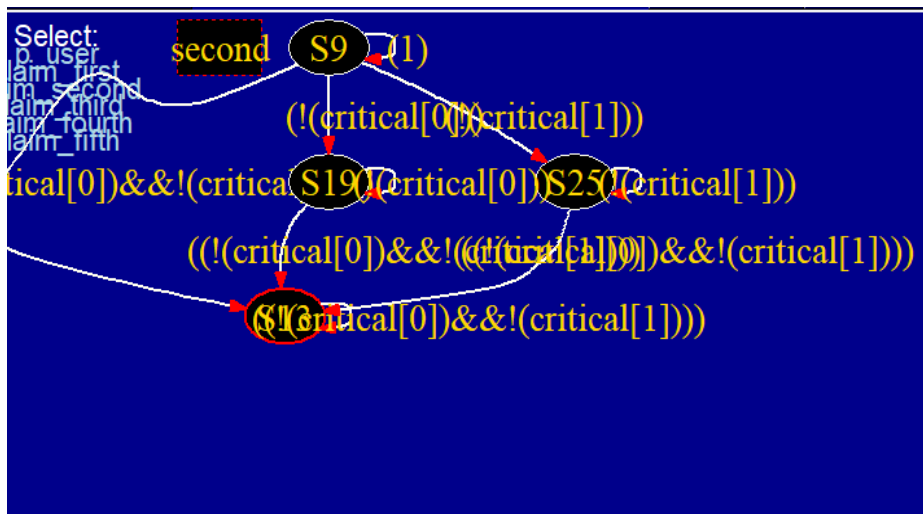
1. ltl first { [] (!critical[0] || !critical[1]) }

This is a safety property that avoids mutual exclusion violation. This means that always, only one process is in the critical section, not both. In this way, it is being ensured that there is no mutual exclusion violation happening and it ran perfectly, leaving no errors. The state diagram obtained is as follows :



2. $\text{ltl second } \{ [] \langle \rangle (\text{critical}[0]) \mid \mid [] \langle \rangle (\text{critical}[1]) \}$

This property says that, no matter what, always a process will eventually get to a critical section. Here, `critical[0]` means that process 0 is in the critical state. The specification holds good and there are no errors or assertions violated during the entire course of the program. The state diagram thus obtained is :



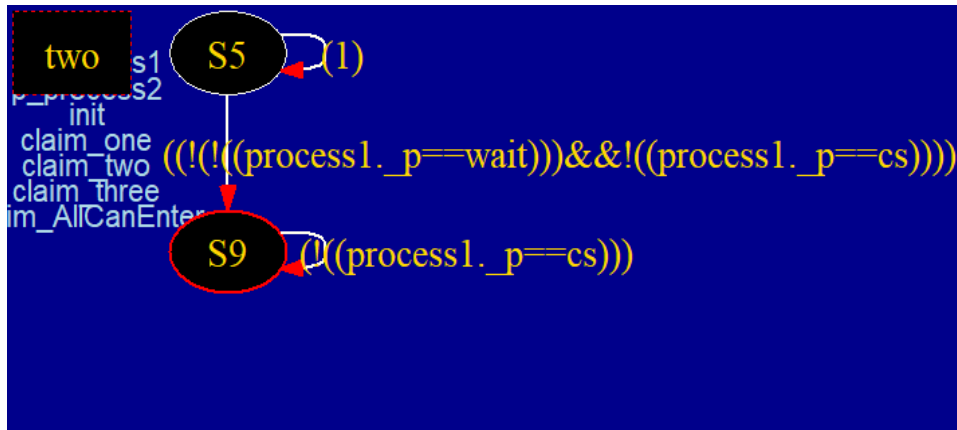
3. $\text{ltl third } \{ [] \langle \rangle (\text{critical}[0] \rightarrow \text{critical}[1]) \mid \mid (\text{critical}[1] \rightarrow \text{critical}[0]) \}$

This property defines liveness, where, a process is not always stuck in one state and constantly keeps moving to another. Here it means that, always, a process eventually moves out of the critical section and never experiences deadlock or starvation.

2. $\text{ltl two } \{ [] (\text{wait1} \rightarrow (<>(\text{cs1}))) \}$

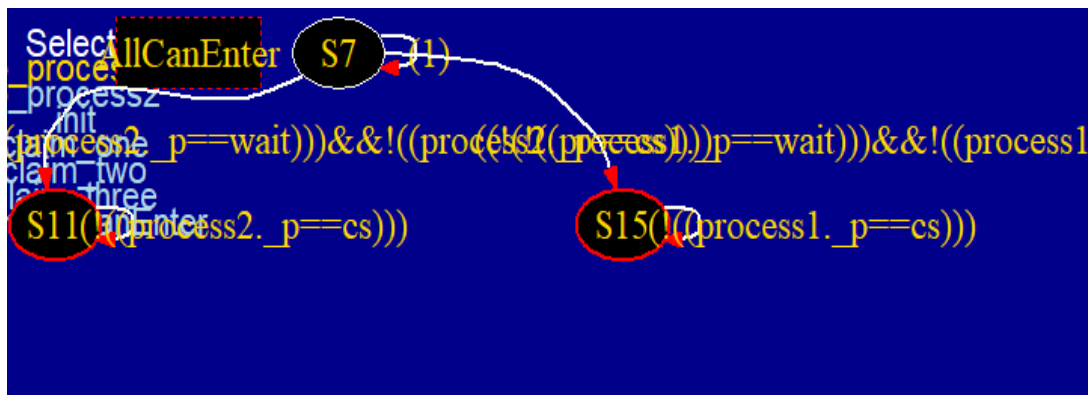
This property talks about the liveness where the process avoids starvation. It means that always, whenever a process is waiting for its turn, it eventually gets to be in the critical section. The same property can be extended to the second process as well by writing :

$\text{ltl three } \{ [] (\text{wait2} \rightarrow (<>(\text{cs2}))) \}$



3. $\text{ltl AllCanEnter } \{ [] ((\text{wait1} \rightarrow (<>(\text{cs1}))) \& \& (\text{wait2} \rightarrow (<>(\text{cs2}))) \}$

This talks about the fairness property where always, both the threads can enter the critical section as mentioned above. The state diagram obtained for this property is shown as below :

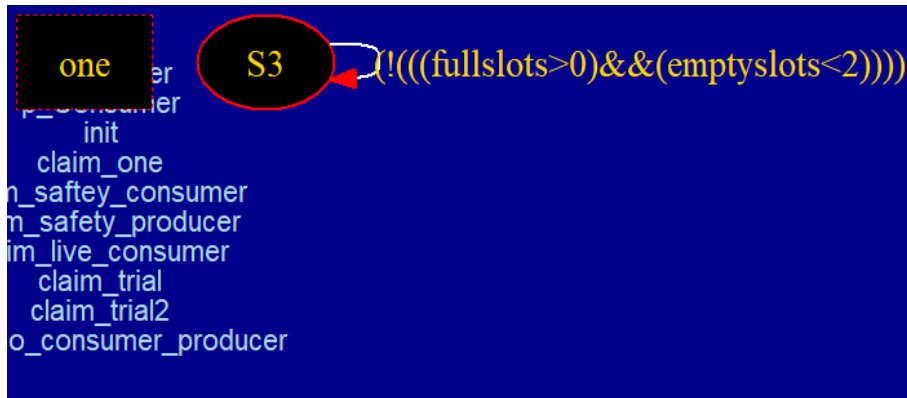


For the **final Producer Consumer Buffer**, to prove the correctness properties, the specifications used are :

1. $\text{ltl one } \{ <> (\text{fullslots} > 0 \& \& \text{emptyslots} < \text{BUFFERSIZE})$

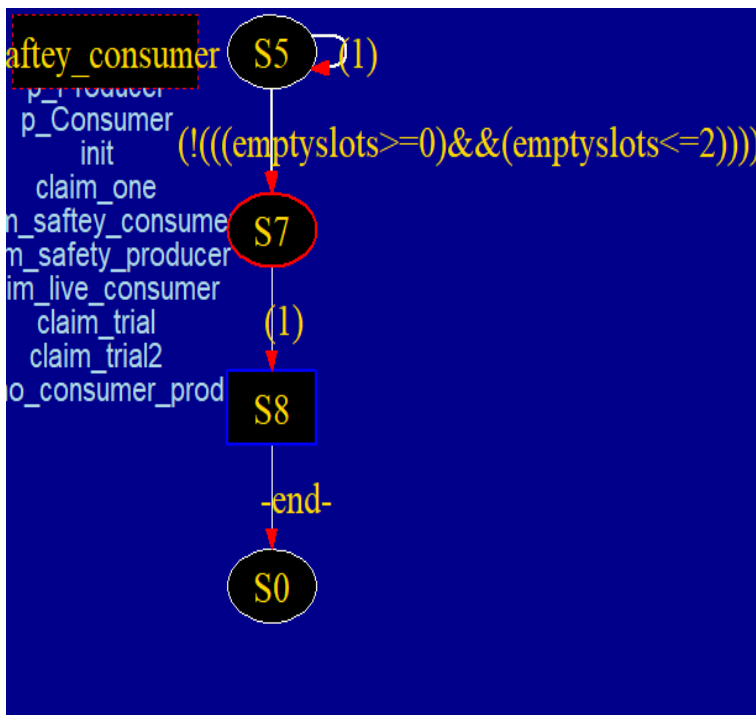
This property talks about the liveness property where the number of full slots and empty slots keep changing with time. The reason for the boundary is that initially fullslots were initialized to

0 and empty slots were initialised to the size of the buffer. The property got executed without any errors and the following state diagram was observed.



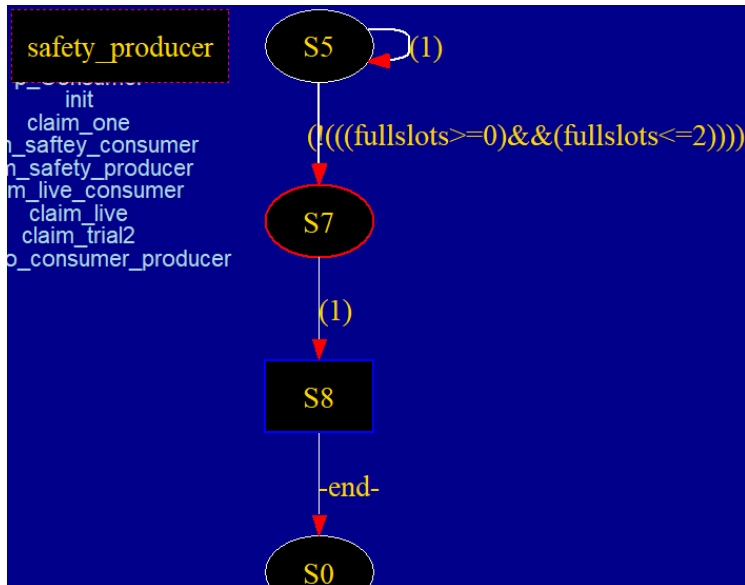
2. `ltl safety_consumer { [] (emptyslots >= 0 && emptyslots <= BUFFERSIZE) }`

This property explains a safety property where `emptyslots` are always greater than or equal to 0 but less than or equal to `buffersize`. In this way we can ensure the fundamental property where the consumer can't take out any more packets from an empty buffer.



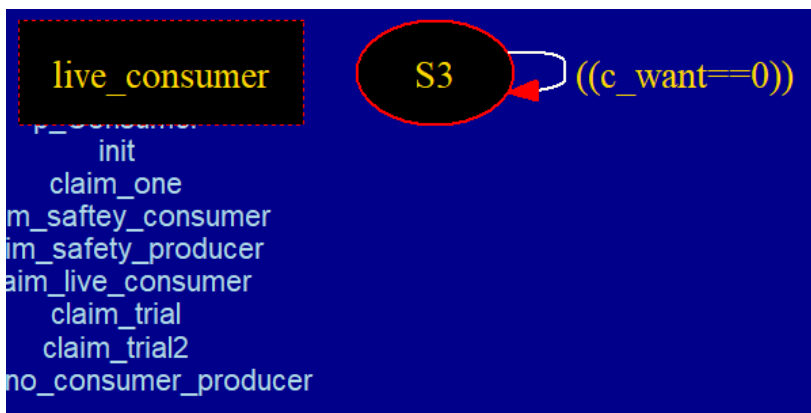
3. `ltl safety_producer { [] (fullslots >= 0 && fullslots <= BUFFERSIZE) }`

This property is similar to the one explained above. Similarly, with this property, we make sure that the producer doesn't produce add to the already complete or filled buffer. Thus enabling this property maintains the safety condition. The following state diagram is obtained



4. $\text{ltl live_consumer } \{ \neg ([] (c_want == 0)) \}$

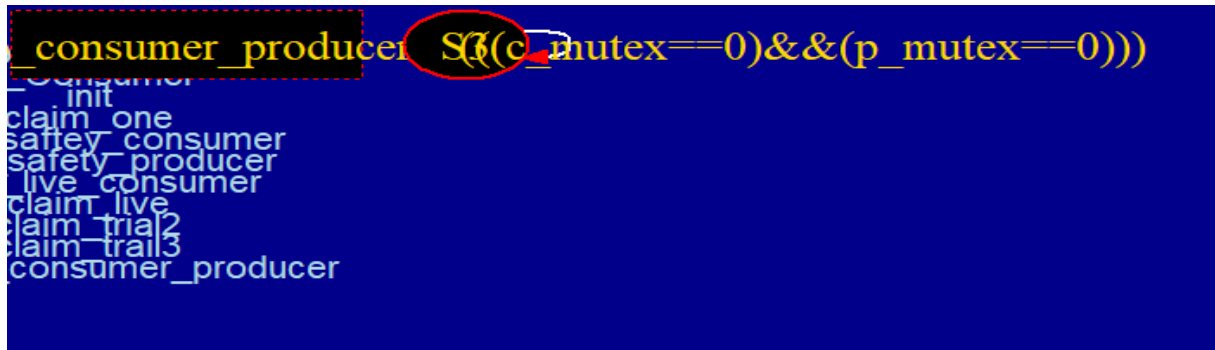
This is another liveness property where we make sure that the consumer always doesn't stay in one state and goes to active state at some point. The above property helps in verifying the algorithm and produces the below state diagram.



Similarly, we can also write : $\text{ltl live_producer } \{ \neg (< (p_want > 0)) \}$

5. $\text{ltl no_consumer_producer } \{ \neg ([] (c_mutex == 0 \& \& p_mutex == 0)) \}$

The property of mutual exclusion is being verified here, where at all times, not both the processes can exist in the critical section at the same time. This verifies the safety and mutual exclusion property. The following state diagram is obtained and didn't produce any errors during verification :



Developing the Application and Specification :

My base application for this project was Mutual Exclusion using Peterson's algorithm. The promela model is then generated from the initial C code through modex. After getting the Promela code file, I analysed the state diagram obtained from it and calculated how the relevant specifications would resemble like. Accordingly, I gave the properties for liveness, safety and fairness so that the correctness of the system could be verified. For example, for fairness, I validated my model with $\text{ltl third } \{ [] ((\text{critical}[0] \rightarrow \text{critical}[1]) \mid \mid (\text{critical}[1] \rightarrow \text{critical}[0])) \}$. Here it means that, both the processes get chance to be in the critical section, and there won't be any violations of mutual exclusion or deadlocks.

After writing the properties, they were made simulated and verified against all the correctness properties. Since, it gave me valid state diagrams and no errors were produced, I went on to build the next stage of the process. Before moving on to the Producer Consumer buffer, to familiarize myself more with this concept, I implemented another model, yet a bit more complex model for mutual exclusion.

The LTL properties thus obtained are :

- $\text{ltl first: } [] ((\neg (\text{critical}[0])) \mid \mid (\neg (\text{critical}[1])))$
- $\text{ltl second: } ([] (<\> (\text{critical}[0]))) \mid \mid ([] (<\> (\text{critical}[1])))$
- $\text{ltl third: } [] (((\neg (\text{critical}[0])) \mid \mid (\text{critical}[1])) \mid \mid ((\neg (\text{critical}[1])) \mid \mid (\text{critical}[0])))$

In the previous model, we can see that mutual exclusion could only happen between two threads, whereas in the second model, we can manually enter the number of threads we want by adding a few additional statements whenever necessary. I learnt how a mutual exclusion model works using “wait” and I implemented a similar model in promela. In this model, each process makes a request saying that they want to enter the critical section and wait for their turn to come. After writing the specifications for the first model, writing the specifications for the second model became much easier.

The development of the specifications was mostly based on the conceptual knowledge of liveness and safety. For example, to prove that if one process wants to enter the critical section, at somepoint of the time, it has to enter and not be starved. This could easily be written as $\{ [] (\text{wait1} \rightarrow (<>(\text{cs1}))) \}$. In this way, as explained in the above section, further more properties are developed to prove the correctness of the system.

The LTL properties thus obtained are :

- $\text{ltl one } \{ [] (! (\text{cs1} \ \&\& \ \text{cs2})) \}$
- $\text{ltl two } \{ [] (\text{wait1} \rightarrow (<>(\text{cs1}))) \}$
- $\text{ltl three } \{ [] (\text{wait2} \rightarrow (<>(\text{cs2}))) \}$
- $\text{ltl AllCanEnter } \{ [] ((\text{wait1} \rightarrow (<> (\text{cs1}))) \ \&\& \ (\text{wait2} \rightarrow (<> (\text{cs2})))) \}$

After this, from the guidance obtained in the class, I tried implementing the Producer Consumer buffer application using mutual exclusion. the producer–consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. This would become our safety condition while writing the specifications. The algorithm implemented could be applied for multiple processes as well.

When verified and simulated, the code ran without running into any errors and gave appropriate state diagrams for the respective functions as well. After that, I started writing my specifications in Ltl to verify its correctness. One property that I knew from the code should always be satisfied is that, the number of emptyslots(which were initially of the buffer size), should always only lie between 0 and the Max size of the buffer. Using this logic, this could be extended to the full slots as well, ie, the number of full slots also must always lie in between the range of 0 and the buffer size. This became another safety property. To assert the liveness of the system , it was also made

sure that the processes don't always stay in the critical section or out of it and correctness was verified accordingly. In this way, from the basic model, all the algorithms and the specifications are developed accordingly.

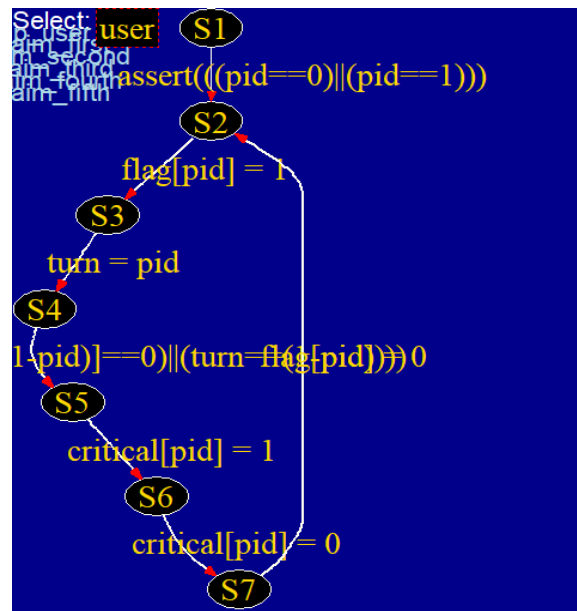
The LTL properties thus obtained are :

- $\text{ltl one: } \langle \rangle (((\text{fullslots} > 0)) \ \&\& \ ((\text{emptyslots} < \text{BUFFERSIZE})))$
- $\text{ltl safety_consumer: } [] (((\text{emptyslots} \geq 0)) \ \&\& \ ((\text{emptyslots} \leq \text{BUFFERSIZE})))$
- $\text{ltl safety_producer: } [] (((\text{fullslots} \geq 0)) \ \&\& \ ((\text{fullslots} \leq \text{BUFFERSIZE})))$
- $\text{ltl live_producer: } \langle \rangle ((\text{p_want} > 0))$
- $\text{ltl live_consumer: } ! \ ([] ((\text{c_want} == 0)))$
- $\text{ltl live: } \langle \rangle ((! ((\text{c_mutex} == 0))) \ || \ ((\text{c_want} == 0)))$
- $\text{ltl trial2: } \langle \rangle ((! ((\text{c_mutex} == 0))) \ || \ ((\text{c_mutex} == 1)))$
- $\text{ltl no_consumer_producer: } ! \ ([] (((\text{c_mutex} == 0)) \ \&\& \ ((\text{p_mutex} == 0))))$

Model Checking Results and Time Space Usage :

Basic Peterson's Algorithm :

Here, the Peterson's Algorithm for mutual exclusion runs without any errors and the assertion is always maintained true. The state diagram obtained for this algorithm is :



When its sent for verification, the results obtained are :


```
spin -a peterson.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000 -n
Pid: 32784
```

```
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction
```

```
Full statespace search for:
  never claim      - (not selected)
  assertion violations +
  cycle checks     - (disabled by -DSAFETY)
  invalid end states +
```

```
State-vector 20 byte, depth reached 21, errors: 0
  34 states, stored
  23 states, matched
  57 transitions (= stored+matched)
  0 atomic steps
hash conflicts:    0 (resolved)
```

```
  0 atomic steps
hash conflicts:    0 (resolved)

  2.501  memory usage (Mbyte)
```

```
pan: elapsed time 0.001 seconds
No errors found -- did you verify all claims?
```

This process is completed in 34 states and the maximum depth reached by it is 21. It takes a total of 2.501 Mb and the time taken to compute this is 0.001 seconds.

As this is a mutual exclusion example, Spin is informing that the proctype has an unreachable state in the end, which is true since they consist of endless loops. Since, it doesn't provide any error, or generate any counter example, all the properties for mutual exclusion are verified. Moreover, the search is done for invalid endstates as well, and there was no error found. The result for this with the inclusion of unreachable states is obtained as below :

```

Full statespace search for:
  never claim      - (not selected)
  assertion violations +
  cycle checks     - (disabled by -DSAFETY)
  invalid end states +

```

```

State-vector 20 byte, depth reached 21, errors: 0

```

```

  34 states, stored
  23 states, matched
  57 transitions (= stored+matched)
  0 atomic steps

```

```

hash conflicts:      0 (resolved)

```

```

  2.501 memory usage (Mbyte)

```

```

unreached in proctype user
  peterson.pml:22, state 9, "-end-"
  (1 of 9 states)

```

```

pan: elapsed time 0.001 seconds
No errors found -- did you verify all claims?

```

This has show the similar results as the above example and ran for about 0.001 seconds, occupying 2,5 Mb of memory.

When an LTL property like, `ltl first { [] (!critical[0] || !critical[1]) }` is used, the result obtained is :

claim name (opt): first
☐ report unreachable code

Run

Stop

Save Result in:

pan.out

exclusion problem - 1981 */

```

)
0 || _pid == 1);

== 0 || turn == 1 - _pid);

1;
tical section */
0;

```

```

acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 43, errors: 0
  34 states, stored
  23 states, matched
  57 transitions (= stored+matched)
  0 atomic steps
hash conflicts:      0 (resolved)

Stats on memory usage (in Megabytes):
  0.001 equivalent memory usage for states (stored*(State-vector + overhead))
  0.292 actual memory usage for states (unsuccessful compression: 20435.83%)
        state-vector as stored = 8976 byte + 16 byte overhead
  2.000 memory used for hash table (-w19)
  0.305 memory used for DFS stack (-m10000)
  2.501 total actual memory usage

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?

```

Here, when checked for a claim individually, for safety and liveness properties, it returned no error and it used a total memory of 2.501 Mb and took 0seconds to run.

If there is a mistake in the code, an error is produced, ie, Spin returns an error and also produces an error trail and counter example for us to use.

```

assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 15, errors: 1
  8 states, stored
  0 states, matched
  8 transitions (= stored+matched)
  0 atomic steps
hash conflicts:      0 (resolved)

Stats on memory usage (in Megabytes):
  0.000 equivalent memory usage for states (stored*(State-vector + overhead))
  0.292 actual memory usage for states (unsuccessful compression: 86852.27%)
        state-vector as stored = 38199 byte + 16 byte overhead
  2.000 memory used for hash table (-w19)
  0.305 memory used for DFS stack (-m10000)
  2.501 total actual memory usage

pan: elapsed time 0.005 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"

```

```

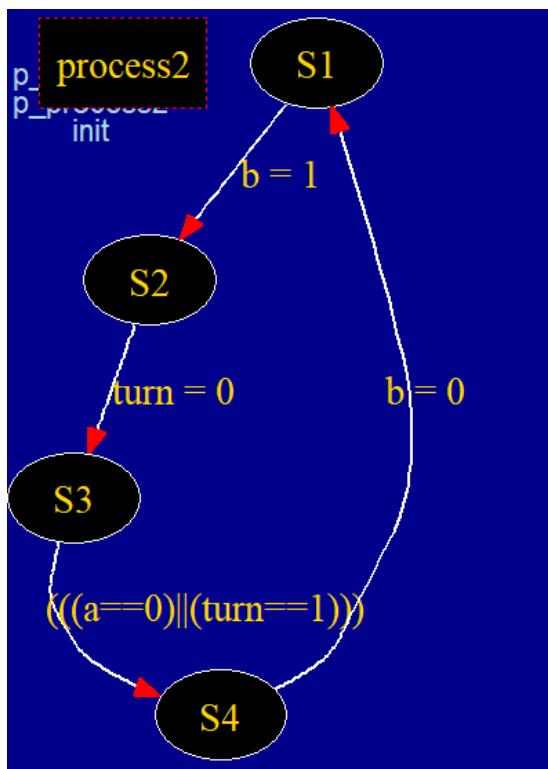
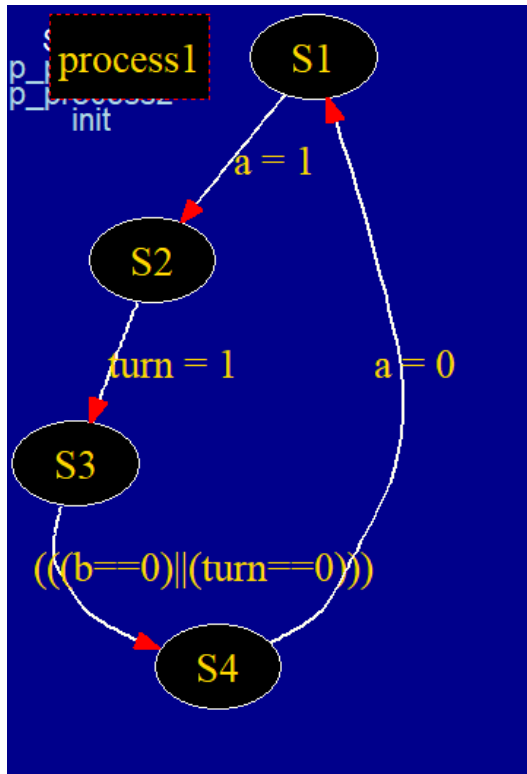
ltl first: [] ((! (critical[1])) || (! (critical[1])))
spin: couldn't find claim 1 (ignored)
2:   proc 1 (user) peterson.pml:10 (state 1) [assert(((pid==0)|| (pid==1)))]
4:   proc 0 (user) peterson.pml:10 (state 1) [assert(((pid==0)|| (pid==1)))]
6:   proc 1 (user) peterson.pml:12 (state 2) [flag[pid] = 1]
8:   proc 1 (user) peterson.pml:13 (state 3) [turn = pid]
10:  proc 1 (user) peterson.pml:14 (state 4) [(((flag[(1-pid)]==0)|| (turn==(1-pid))))]
12:  proc 1 (user) peterson.pml:16 (state 5) [critical[pid] = 1]
14:  proc 1 (user) peterson.pml:18 (state 6) [critical[pid] = 0]
spin: trail ends after 15 steps
#processes: 2
15:  proc 1 (user) peterson.pml:20 (state 7)
15:  proc 0 (user) peterson.pml:12 (state 2)

```

Using the simulation section and guided error trail, we can find our bug as mentioned above and correct it which would lead us to correct implantation of the Ltl example discussed above.

(ii)

For the modified version of the mutual exclusion, the state diagram thus obtained for process 1 and 2 are :



Here we see that the algorithm ran for 22 states, and there were no errors during its verification. When the model is set up for execution, it took up 2.5 Mb of memory, about 0.002 seconds to computer

```
Full statespace search for:
    never claim      - (not selected)
    assertion violations +
    cycle checks     - (disabled by -DSAFETY)
    invalid end states +

State-vector 20 byte, depth reached 16, errors: 0
    22 states, stored
    15 states, matched
    37 transitions (= stored+matched)
    0 atomic steps
hash conflicts:      0 (resolved)

    2.501  memory usage (Mbyte)

unreached in proctype process1
    mutex_ppt.txt:15, state 6, "-end-"
    (1 of 6 states)
unreached in proctype process2
    mutex_ppt.txt:21, state 6, "-end-"
    (1 of 6 states)
(0 of 5 states)

pan: elapsed time 0.002 seconds
No errors found -- did you verify all claims?
```

Here, the process is completed in 22 states with the depth of 2. It totally takes memory of 2.501 Mb and takes about 0.002 seconds to complete the execution.

When the algorithm is tested with the LTL property, “ **ltl two { [] (wait1 -> (<>(cs1))) }** ”, the following result is obtained :

```

acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 33, errors: 0
  28 states, stored (34 visited)
  23 states, matched
  57 transitions (= visited+matched)
  0 atomic steps
hash conflicts:      0 (resolved)

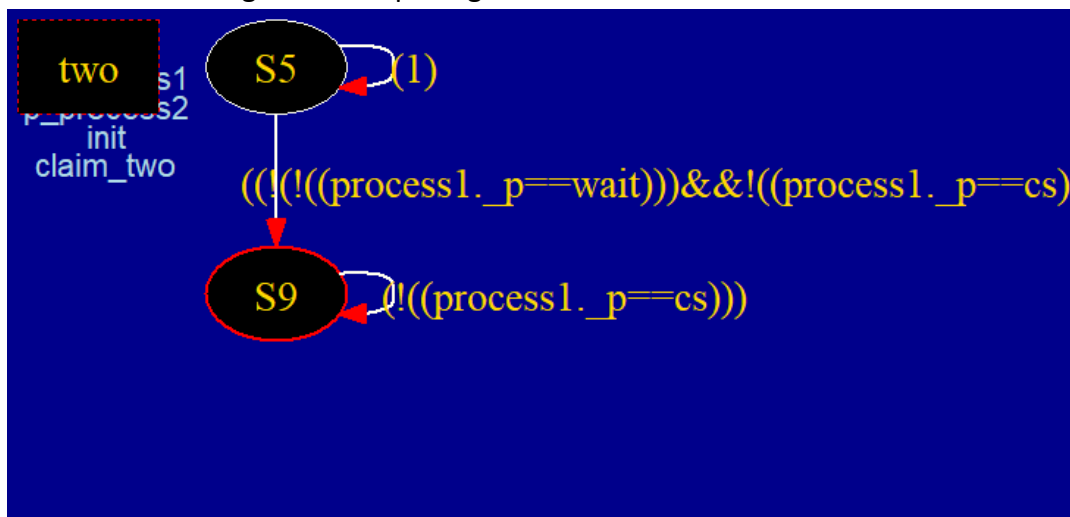
Stats on memory usage (in Megabytes):
  0.001 equivalent memory usage for states (stored*(State-vector + overhead))
  0.291 actual memory usage for states (unsuccessful compression: 24759.74%)
        state-vector as stored = 10878 byte + 16 byte overhead
  2.000 memory used for hash table (-w19)
  0.305 memory used for DFS stack (-m10000)
  2.501 total actual memory usage

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?

```

In this case, where the liveness property of the Model is verified, we observe that there are no errors produced in this case and takes about 2.5 Mb and 0 seconds to run.

The state diagrams depicting the flow is also obtained for this is__:



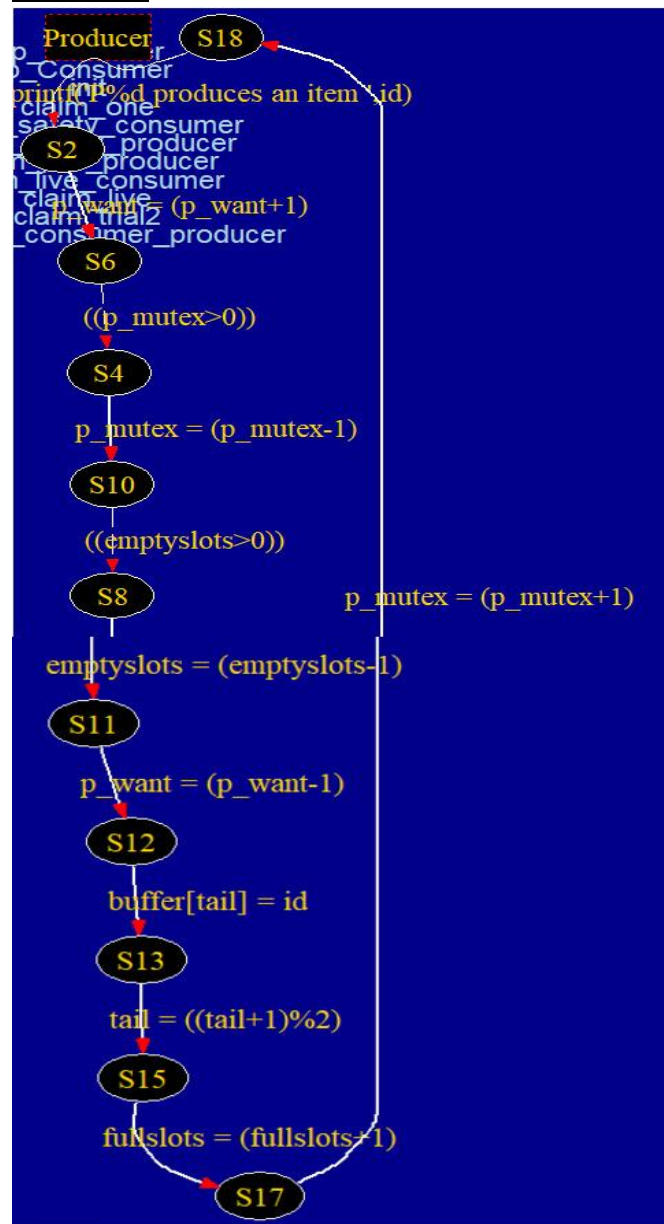
Final Producer and Consumer Buffer :

The Producer and Consumer Application on Mutual Exclusion is the final model obtained. The state automata obtained for both Producer and Consumer buffer is shown as below :

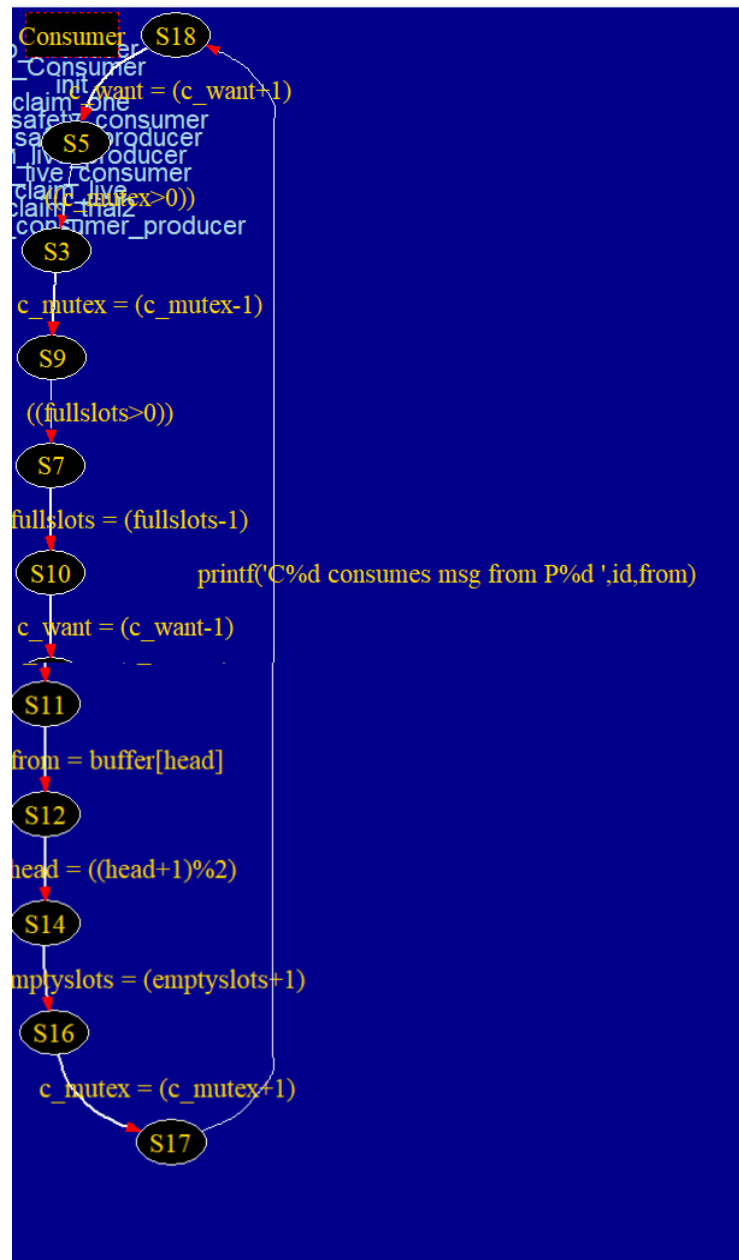
```
graph TD
    Start([Producer]) -- "p_mutex = (p_mutex-1)" --> S2((S2))
    S2 -- "claim one" --> S6((S6))
    S6 -- "wait" --> S4((S4))
    S4 -- "p_mutex = (p_mutex-1)" --> S10((S10))
    S10 -- "wait" --> S8((S8))
    S8 -- "emptyslots = (emptyslots-1)" --> S11((S11))
    S11 -- "p_want = (p_want-1)" --> S12((S12))
    S12 -- "buffer[tail] = id" --> S13((S13))
    S13 -- "tail = ((tail+1)%2)" --> S15((S15))
    S15 -- "fullslots = (fullslots+1)" --> S17((S17))
    S17 -- "p_mutex = (p_mutex+1)" --> S18((S18))
    S18 -- "release" --> Start
```

The flowchart illustrates the execution of a Producer-Consumer problem. The flow starts at a 'Producer' node, proceeds through states S2, S6, S4, S10, S8, S11, S12, S13, S15, and S17, and finally loops back to S18. Each state transition is labeled with a semaphore operation or a condition check.

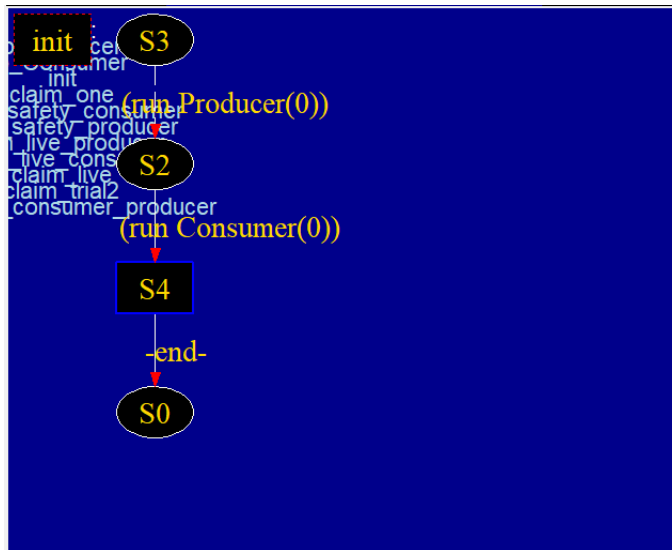
- Producer** (Start) → **S2**: $p_mutex = (p_mutex - 1)$
- S2** → **S6**: **claim one**
- S6** → **S4**: **wait**
- S4** → **S10**: $p_mutex = (p_mutex - 1)$
- S10** → **S8**: **wait**
- S8** → **S11**: $emptyslots = (emptyslots - 1)$
- S11** → **S12**: $p_want = (p_want - 1)$
- S12** → **S13**: $buffer[tail] = id$
- S13** → **S15**: $tail = ((tail + 1) \% 2)$
- S15** → **S17**: $fullslots = (fullslots + 1)$
- S17** → **S18**: $p_mutex = (p_mutex + 1)$
- S18** → **Producer**: **release**



Consumer :



Initialization Process :



The main Producer Consumer Buffer Problem runs without giving any errors and satisfies the correctness, fairness and safety properties as described above. When the buffer size is 2, and one producer and consumer process are used, the following results are obtained in the verification section.

Finite number of states are produced for it as follows. This producer consumer application has 5770 states stored and occupies a memory of 2.794 Mb. The total time taken for its execution is 0.004 seconds.

```
Partial Order Reduction
Full statespace search for:
  never claim      - (not selected)
  assertion violations +
  cycle checks     - (disabled by -DSAFETY)
  invalid end states +

State-vector 44 byte, depth reached 2692, errors: 0
  5770 states, stored
  11058 states, matched
  16828 transitions (= stored+matched)
  1 atomic steps
hash conflicts:    26 (resolved)

  2.794  memory usage (Mbyte)

span: elapsed time 0.004 seconds
No errors found -- did you verify all claims?
```

The results obtained when we decide to see the unreachable states as well are:

```
State-vector 44 byte, depth reached 2692, errors: 0
  5770 states, stored
 11058 states, matched
16828 transitions (= stored+matched)
   1 atomic steps
hash conflicts:    26 (resolved)

  2.794  memory usage (Mbyte)

unreached in proctype Producer
  pc2.txt:53, state 21, "-end-"
  (1 of 21 states)
unreached in proctype Consumer
  pc2.txt:79, state 21, "-end-"
  (1 of 21 states)
unreached in init
  (0 of 4 states)

pan: elapsed time 0.004 seconds
No errors found -- did you verify all claims?
```

The results obtained when two producers and two consumers are running are :

```
Full statespace search for:
  never claim      - (not selected)
  assertion violations +
  cycle checks     - (disabled by -DSAFETY)
  invalid end states +

State-vector 56 byte, depth reached 9999, errors: 0
 43743 states, stored
110681 states, matched
154424 transitions (= stored+matched)
   1 atomic steps
hash conflicts:   2837 (resolved)

  5.333  memory usage (Mbyte)

pan: elapsed time 0.032 seconds
No errors found -- did you verify all claims?
```

Here the total time taken is 0.032 seconds and the total memory occupied is 5.33 Mb

Results obtained when we are checking for one ltl property : “ safety_consumer “

claim name (opt):safety_consumer

☒ report unreachable code

Run

Stop

Save Result in:

pan.out

{process id and a rotating value up to

}

never claim +

assertion violations + (if within scope of claim)

acceptance cycles + (fairness disabled)

invalid end states - (disabled by never claim)

State-vector 60 byte, depth reached 5164, errors: 0

5825 states, stored

11544 states, matched

17369 transitions (= stored+matched)

1 atomic steps

hash conflicts: 49 (resolved)

Stats on memory usage (in Megabytes):

0.422 equivalent memory usage for states (stored*(State-vector + overhead))

0.577 actual memory usage for states (unsuccessful compression: 136.75%)

state-vector as stored = 88 byte + 16 byte overhead

2.000 memory used for hash table (-w19)

0.305 memory used for DFS stack (-m10000)

2.794 total actual memory usage

pan: elapsed time 0.005 seconds

For checking this property, it took a total of 5825 states and 2.794 Mb. This is done when only 1 producer and 1 consumer are used. Hence in this way we can verify that, even if the input is changed a little, the correctness properties of the algorithm are still maintained. In the last section, the results including the space and time usage are also explained.