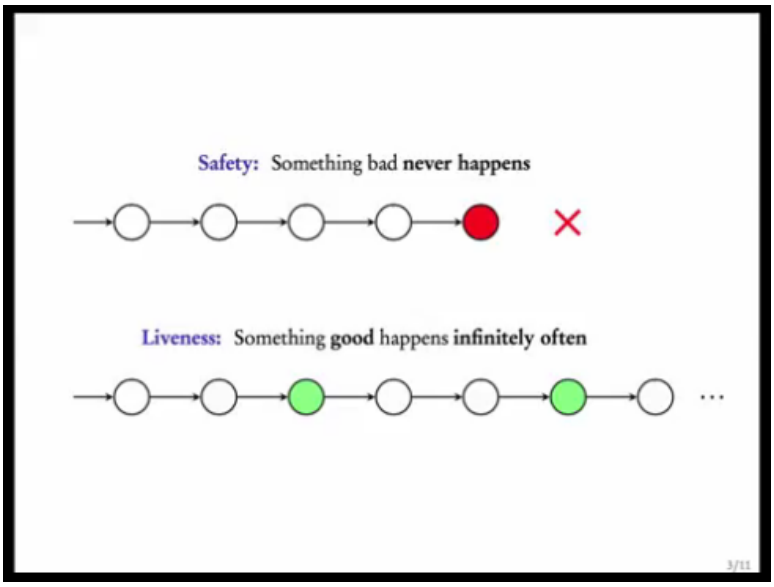
Model Checking
Prof. B. Srivathsan
Department of Computer Science and Engineering
Indian Institute of Technology – Madras

Lecture - 16
Liveness Properties

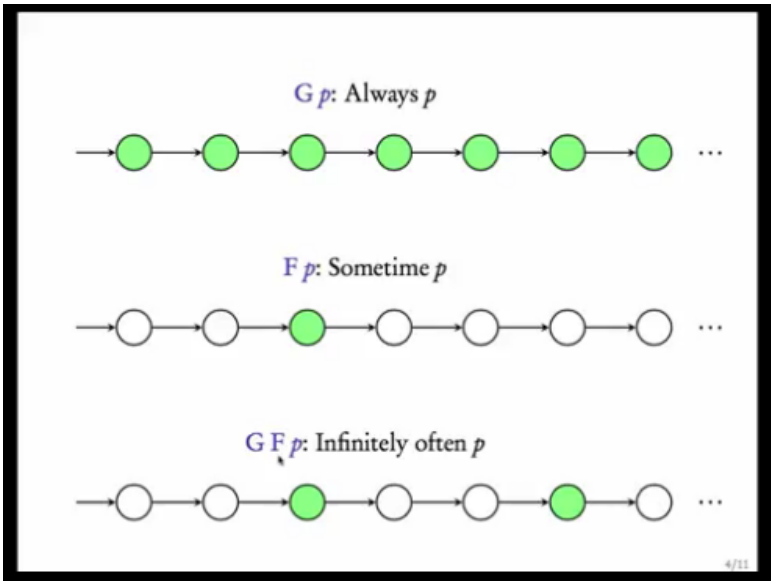
In the last module we saw a kind of property called safety property. Safety properties were useful to say that something bad does not happen. In this module we will see what are called liveness properties.

(Refer Slide Time: 00:21)



As I said a safety property ensures that something bad never happens. In particular for a safety property there is a set of bad prefixes that it avoids. Liveness property comes with a different flavor a liveness property says that something good happens infinitely often. Here for example the good is the green state the liveness says that the green state appears infinitely often we will see examples.

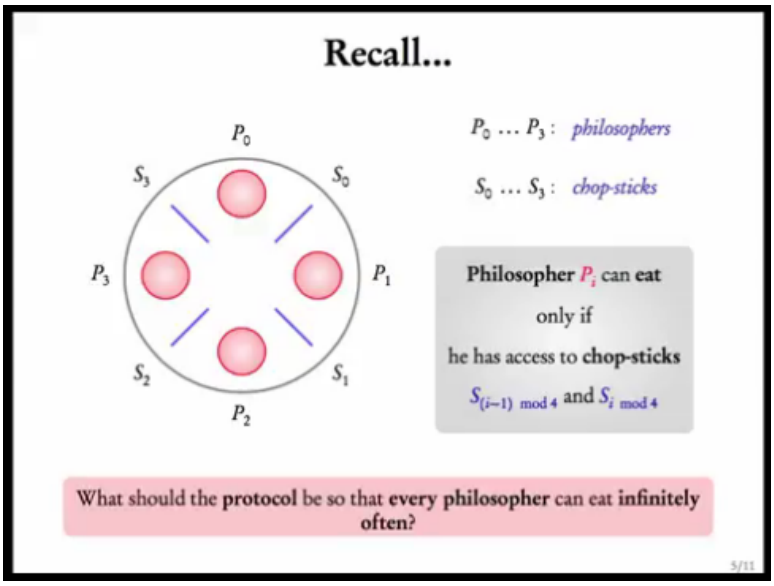
(Refer Slide Time: 01:04)



Let us see how do we specify liveness properties first. When we say G p that means p should be always true this is always p. When we say F p sometime during the execution p is true. How do we say infinitely often something is true not necessarily everywhere but not just once but infinitely often like this.

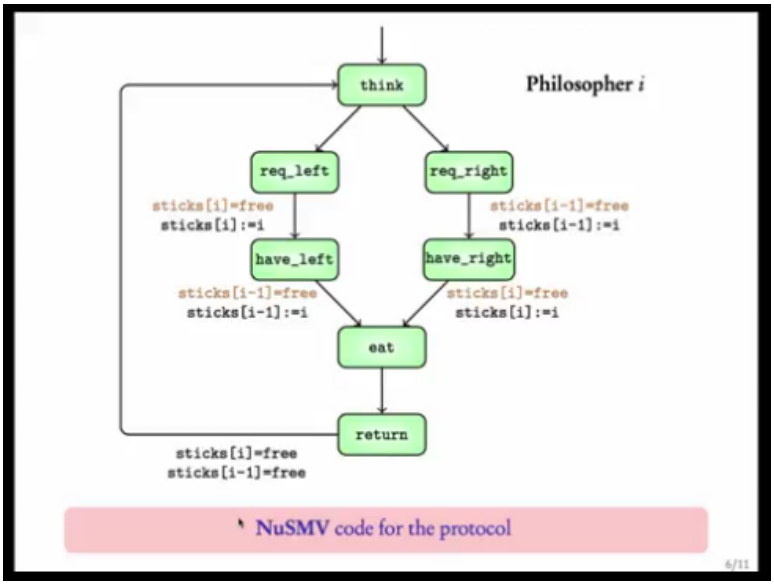
To specify a property like that you can combine G F to say G F p this just says that in every state there is a possibility of going to p. So see F p says that from the initial state sometimes you reach p. G F p says that from every state F p is true that means from every state there is a future state where p is true that means infinitely often p. So, here F p is true, here F p is true because of this, here F p is true because of this state itself, here F p is true because of this state, here F p is true because of this state and so on.

(Refer Slide Time: 02:38)

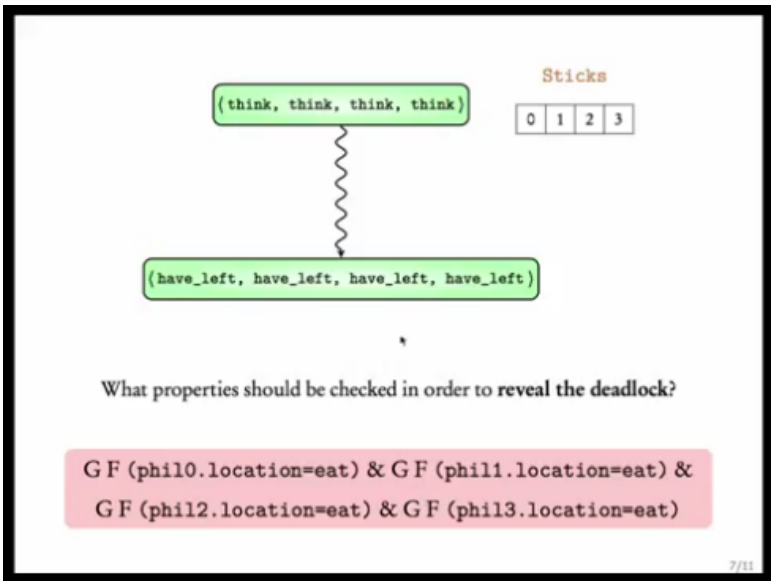


How do we use G F? Let us now finally recall the dining philosophers problem with which we started in module 1. We had philosophers and sticks a philosopher can eat only if he has access to both the sticks on his sides. The question was what should the protocol be so that every philosopher can eat infinitely often? This is the question with which we started with.

(Refer Slide Time: 03:19)



(Refer Slide Time: 03:31)



We saw an example of a protocol we wrote the protocol in NuSMV. If you remember we had a deadlock scenario from the initial state where every philosopher is in think state there was a situation where each of them went to the have left. Let me recall the transition system representing philosopher initially philosopher is in the think state.

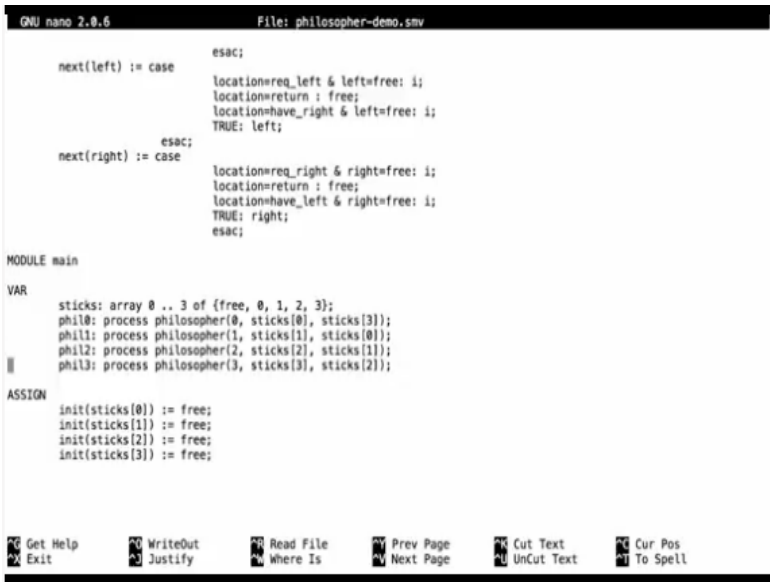
At some point he can non deterministically choose to go either left or right that means he can either request for the left stick or he can make a request for the right stick. If he is here that means if he has requested for the left stick and if indeed the left stick is free then he can have the left stick. In the process he makes an assignment saying that I have the left stick that means sticks of i is equal to philosophers number i.

If he has the left stick he waits for the right stick and if the right stick is free he can go to the eat state. In the process he sets that the right stick is in my possession this is a symmetric path he can eat as long as he wants. And then when he is about to return he goes to the return state from the return state he has to go back to think. In the process he sets both the sticks to be free.

The problem with this protocol was it could lead to deadlock scenario where all of them are having the left stick and waiting for the right stick forever. We identified this deadlock scenario by forcefully simulating certain paths but in general given a transition system

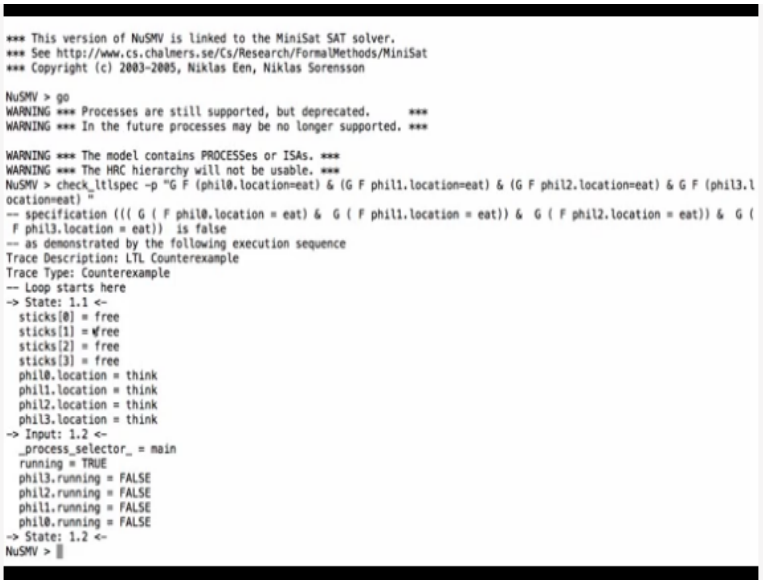
what properties should be checked in order to reveal the deadlock. Here we will make use of liveness; we will check if infinitely often each of the philosophers can eat. Let us see what happens when we checked this condition on this model.

(Refer Slide Time: 06:05)



Here is the philosopher demo dot smv which implements the philosopher transition system that we saw in the slides. These are locations these are the transition edges and in the main we are calling 4 philosophers using the keyword process we saw this code in module 1. Let us now try to check the property if infinitely often each of the philosophers can eat.

(Refer Slide Time: 06:55)

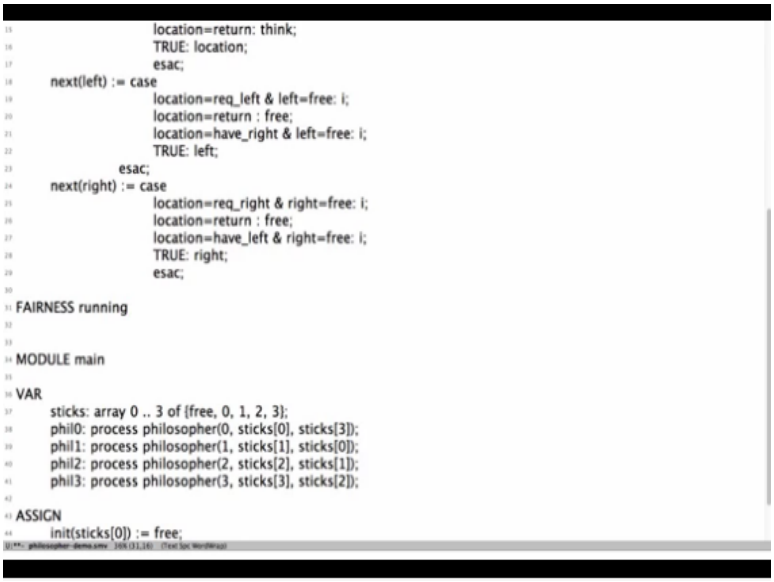


NuSMV, so the specification would be check ltl spec G F philosopher 0 dot location equal to eat and G F philosopher 1 dot location equal to eat and same for the other 2 philosophers. And finally, let us now check the specification it says that the specification is false there is an execution where the philosophers do not get to eat infinitely often.

Let us have a look at the counterexample; this is the initial state now in state 2 so there is a loop here state 2 is the same state but that is because the process selected is the main process. This is a counterexample where the philosopher processors are not even scheduled.

This is a counterexample due to the main process being scheduled. This is a not a satisfactory counterexample because the scheduler is being unfair it is not even scheduling the philosophers. We can try to get rid of such counterexamples by adding the following keyword that I am going to save.

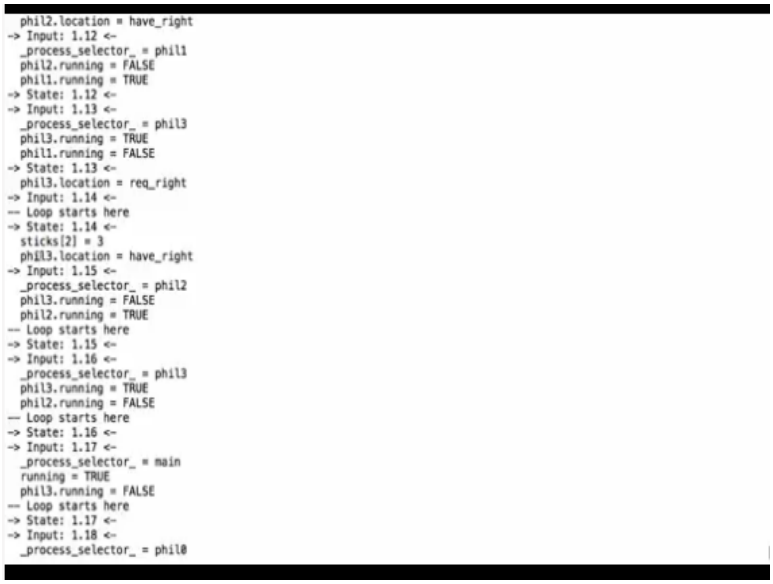
(Refer Slide Time: 09:01)



So, here is the code we want every philosopher to be scheduled infinitely often it should not be the case that the scheduler does not give the philosopher a chance to take the next step. Such a situation where the philosopher is not even given a chance rather in general the process is not even given a chance is called starvation.

We do not want to starve a process so we will add in the module this statement saying fairness running. When we say fairness running the module which contains this keyword is scheduled infinitely often.

(Refer Slide Time: 10:19)



Let us see the effect of this so still there is a counterexample to this property. Let us see what the counterexample is it says that initially all of them are thinking and the sticks are free. Now, the philosopher process 0 is selected and he goes to request right then philosopher 1 is selected.

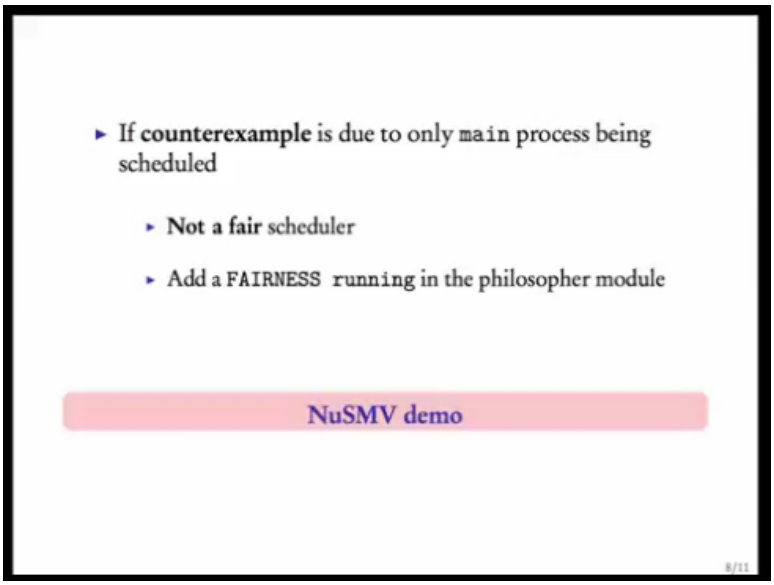
Let us first just look at the locations so philosopher dot look philosopher 0 goes to have right then philosopher 1 requests right he goes to have right, philosopher 2 requests right then philosopher 2 goes to have the right, philosopher3 requests right and then a loop starts where philosopher 3 has right and then there is no change in the state. So, there are multiple loop starts here you should see the 1 in the top most the first place where loop starts here is the beginning of the loop.

Now, all the philosophers are in the have right situation lets see what is there inside the loop. All the processes are being scheduled in the loop philosopher 0,3,2,1 etcetera but there is no change in the state. This is exactly what we wanted a situation where all the philosopher philosophers go to the have right state and all of them are being scheduled.

So, the scheduler is being fair and still there is no change in the state this is representing a deadlock. And how did we detect this? We gave the property is it possible for the philosophers to eat infinitely often. So, here eating is the good part and we asked if this is good thing can happen infinitely often.

In the case of processes execution are going to the critical section would be the good part and we would want to ask if the process can go to the critical section infinitely often. And this model does not let the philosopher eat infinitely often because of this counterexample here where everyone goes to have right and then there cannot be any change in the state this is a kind of counterexample that we wanted.

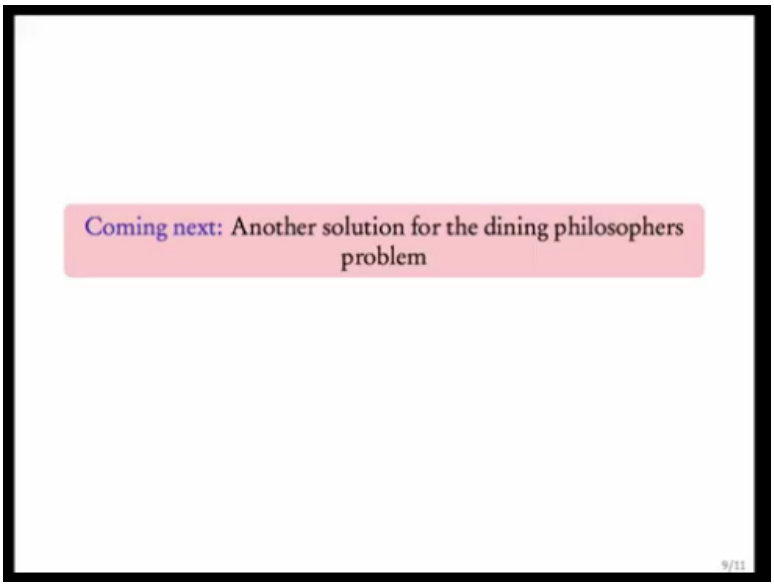
(Refer Slide Time: 13:28)



Let me summarize the part about the counterexample which we saw the first counterexample that we saw was due to only the main process being scheduled and this is not a fair scheduler. We should add fairness running in the philosopher module so each time the module is instantiated the variable or the process that instantiates this module will be scheduled infinitely often.

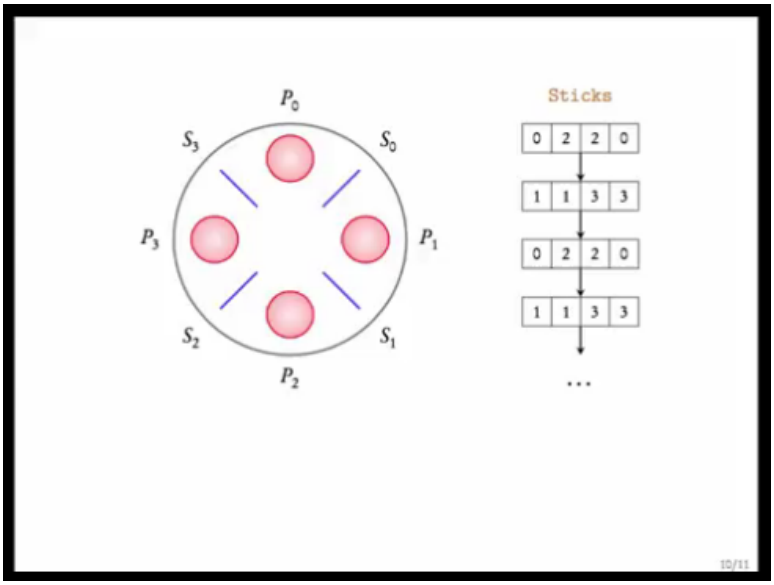
After doing this we got a counterexample which was actually due to a problem with the model and not because of a problem with the main process being scheduled in an unfair manner. We saw a NuSMV demo of this.

(Refer Slide Time: 14:26)



Let us now see another solution to the dinning philosophers problem where this deadlock can be avoided.

(Refer Slide Time: 14:39)



So, here is the scenario we want to prevent the scenario where each philosopher has one stick. How do we prevent this? Initially stick s0 can be accessed by the philosophers to its right, stick 1 by the philosopher to its left which is 2, stick 3 by the philosopher to its right rather sorry stick 2 by the philosopher to its right which is 2 and finally stick 3 by the philosopher to its left which is 0.

What does this allow? This allows philosopher 0 and philosopher 2 to eat after they eat they set the availability of stick 0 to 1, availability of stick 1 again to 1, stick 2 to 3 and stick 3 to 3 this is the scenario. So, essentially stick 0 is initially available to its right and then available to its left. In an alternate way stick 1 is initially available to its left and then available to its right and so on.

So once philosopher 1 and 3 finish eating the sticks accessibility are changed. Stick 0 is with philosopher 1 now what he does is he makes stick 0 available to philosopher 0, stick 1 is also available to philosopher 1 he sets the availability of stick 1 to 2 and so on. This can be continue and alternately philosopher 2 and 0 and 2 and 1 and 3 can be eating. This will avoid a deadlock system scenario. We can see a demo of this in NuSMV now.

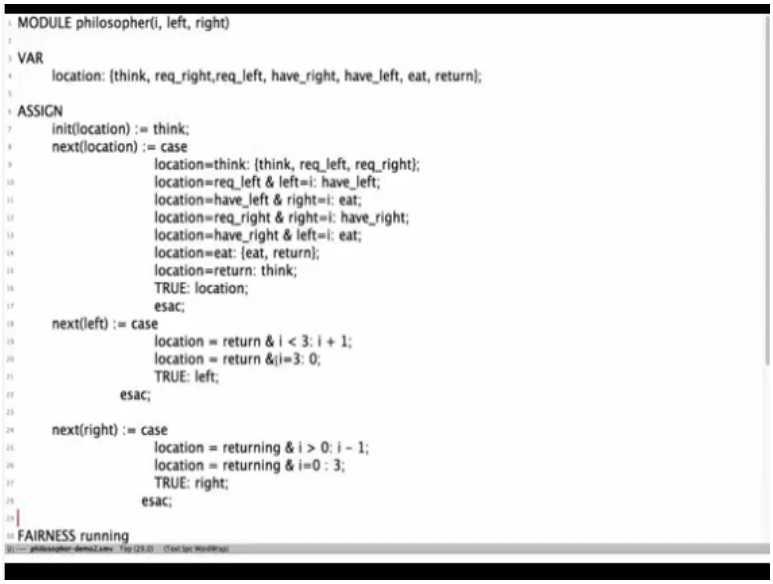
(Refer Slide Time: 16:52)

```
MODULE philosopher(i, left, right)

VAR
        location: {think, req_right,req_left, have_right, have_left, eat, return};

ASSIGN
        init(location) := think;
        next(location) := case
                                location=think: {think, req_left, req_right};
                                location=req_left & left=i: have_left;
                                location=have_left & right=i: eat;
                                location=req_right & right=i: have_right;
                                location=have_right & left=i: eat;
                                location=eat: {eat, return};
                                location=return: think;
                                TRUE: location;
                                esac;
        next(left) := case
                                location = return & i < 3: i + 1;
                                location = return & i=3: 0;
                                TRUE: left;
                        esac;

        next(right) := case
                                location = returning & i > 0: i - 1;
                                location = returning & i=0 : 3;
                                TRUE: right;
                              esac;

FAIRNESS running
```

So, here is the modified code for this new solution let me explain the modifications. Firstly the arrays sticks can take only 0, 1, 2, 3 it cannot take the word free. The initial allocation of the 0 th sticks is with process 0. The first stick goes to 2, the second stick goes to 2 and third stick goes to 0. This is along the same lines as here initially sticks 0 is with 0, 1 is with 2, 2 is with 2 and 3 is with 0. How do we change it? In the next transitions there are some changes as I will describe now.

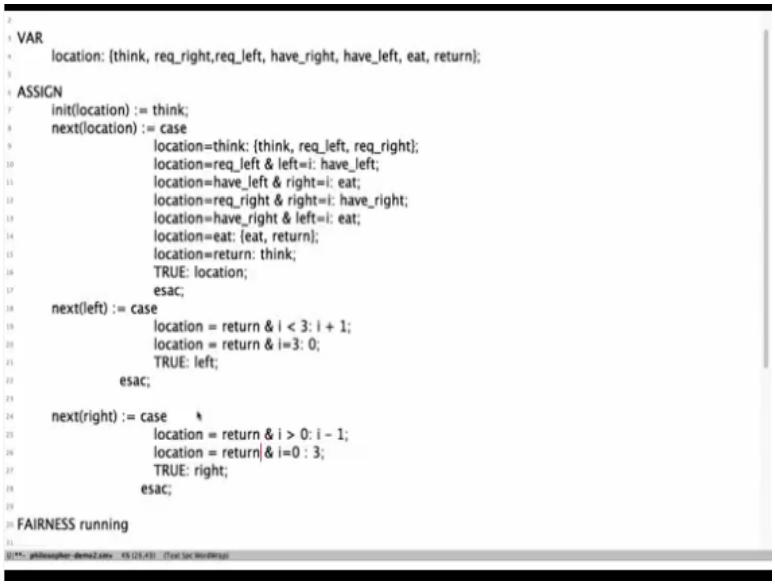
First of all, if the philosopher in his is in his request left location he can have the left stick only if the left stick takes the value i. In the previous case we had set free now it will be i. Similarly, he can have the right stick only if the value of the right is i now these are the changes here. Now, how does the left stick change once he returns the stick.

(Refer Slide Time: 18:38)



Look at philosopher 2 for instance; initially he has position to the left stick and the right stick. The left stick is S2 and the right stick is S1 after he eats he should pass the left stick to P3 and the right stick to P1 we get this. Once he wants to return the left stick is made i plus 1 and the right stick is made i minus 1.

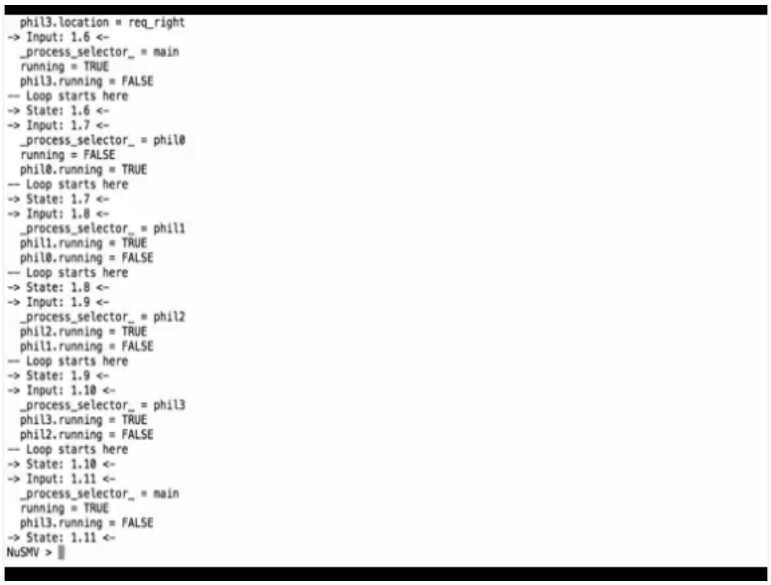
(Refer Slide Time: 19:05)



Just for the special case when i equal to 3 for i equal to 3 his left stick is S3 so once he is done with S3 his left stick should go to 0 not 3 plus 1 its 3 plus 1 modular 4 so that is why we have this statement. If it is the philosopher 3 whose returning then he should return the stick to 0.

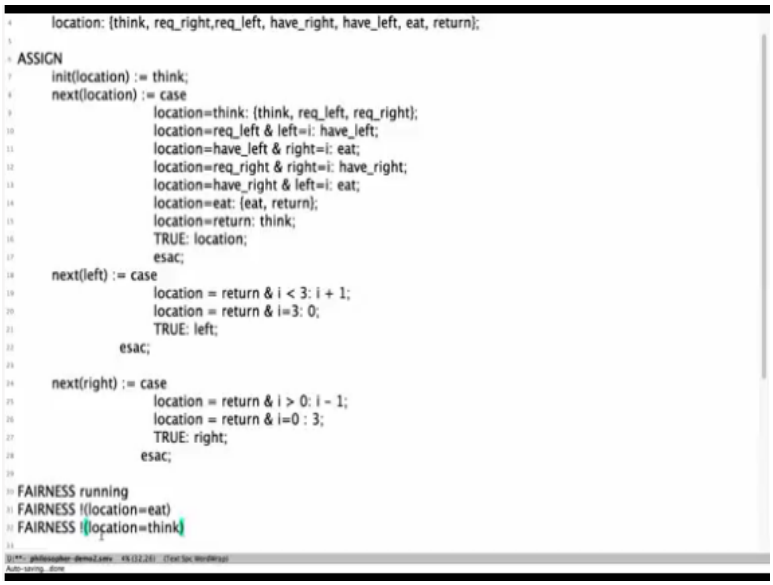
Similarly, the right stick goes to i minus 1 on return so this should be return. Just in a particular case when i is 0 then the right stick goes to 3 this is similar to this scenario. This is the changed philosopher according to the second solution we already have the fairness running here. So we want to see that if the philosophers are executed infinitely often will they get to eat infinitely often let us check this.

(Refer Slide Time: 19:51)



So, here is the specification from the slides we would expect that this specification is true let us check this. However, it says that the specification is false that means there is an execution where the philosophers do not get to eat infinitely often.

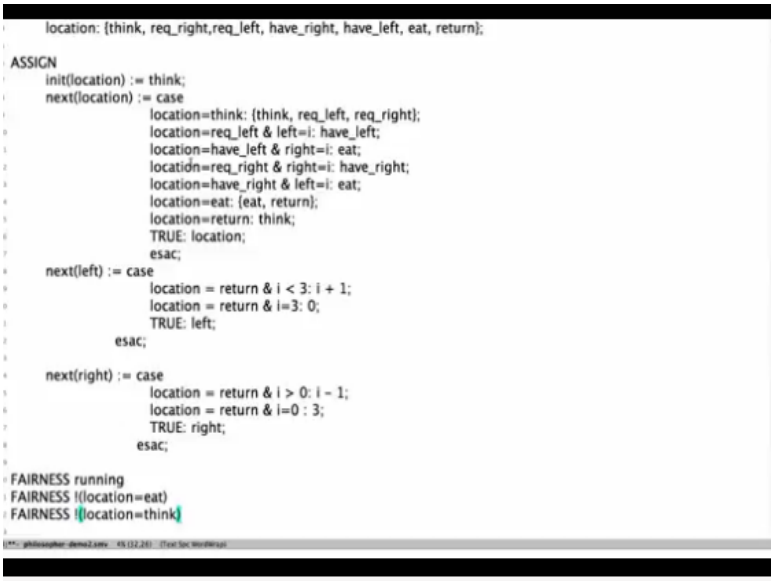
(Refer Slide Time: 20:17)



Let us now check it and see what is the thing that is going wrong? Initially all of them are thinking then philosopher 1 goes to request right, philosopher 3 goes to request right and then a loop starts. There is no change in state that means some philosophers are still thinking. Why does this happen like this? If you remember we had this possibility if a philosopher is in think he can non-deterministically choose to either think or request left or request right no one is prohibiting him from just staying in the think state.

Similarly, if a philosopher is in the eat state he can continue eating we want to avoid the scenarios where the philosopher keeps thinking infinitely often or eating infinitely often. In the sense he keep staying he keeps taking this transition always from eat to eat or from think to think.

(Refer Slide Time: 21:47)



This we can specify by saying that fairness is not the case that location equal to eat and fairness is not the case that location equal to think. This will reduce all the executions where the philosopher does not keep thinking infinitely often in sense he does not keep staying in the think state forever.

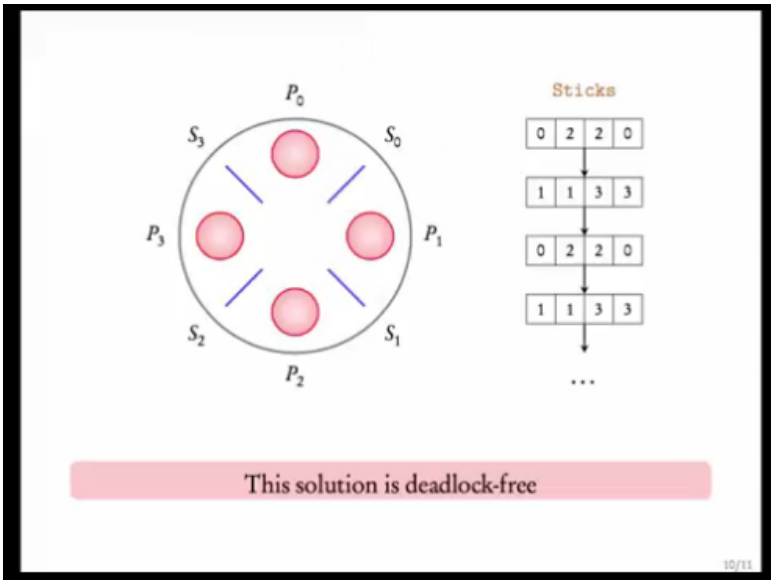
And similarly, he does not keep staying in the eat state forever fairness not of location equal to eat means let us look at the executions where the location is not equal to eat infinitely often. Let us look at the executions where the location is not equal to think infinitely often that means we are looking at executions where the face where the philosopher is thinking or eating is finite. Let us now check the same specification on this modified code.

(Refer Slide Time: 22:52)

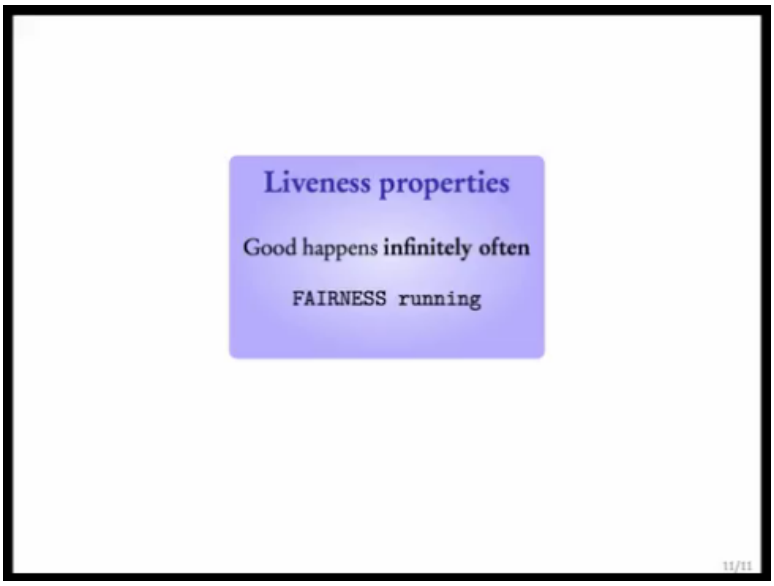
```
srivathsan:NuSMV sri$ NuSMV -int philosopher-demo2.smv
*** This is NuSMV 2.5.4 (compiled on Fri Nov 23 21:36:06 UTC 2012)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

NuSMV > go
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > check_ltlspec -p "(G F phil0.location=eat) & (G F phil1.location=eat) & (G F phil2.location=eat) & (G F phil3.l
ocation=eat)"
-- specification ((( G ( F phil0.location = eat) &  G ( F phil1.location = eat)) &  G ( F phil2.location = eat)) &  G (
 F phil3.location = eat))  is true
NuSMV >
```

Here we are let's check the specification of the modified code and NuSMV says that the specification is true. So along all parts where the philosopher doesn't get stuck in the eat and think state this specification is indeed true. So, if you do not allow them to do these unrealistic things the specification holds and there is no deadlock so this solution is deadlock free.

(Refer Slide Time: 23:24)



(Refer Slide Time: 23:28)



Let us now summarize this module. We introduce what are called liveness properties where you can check if something good happens infinitely often. While doing this we need to take care of fair executions we will not want to take care of executions where the process is not being scheduled at all or the process stays in some state infinitely often just because of the way the transitions are modeled.

If you remember from think we had a possibility of staying in think so it should not be this case which is creating problems. Hence, we gave fairness running and fairness not of location equal to eat and fairness not of location equal to think. This reduce the executions to realistic scenarios where the processes are being scheduled and they do not keep staying in their critical sections forever.