

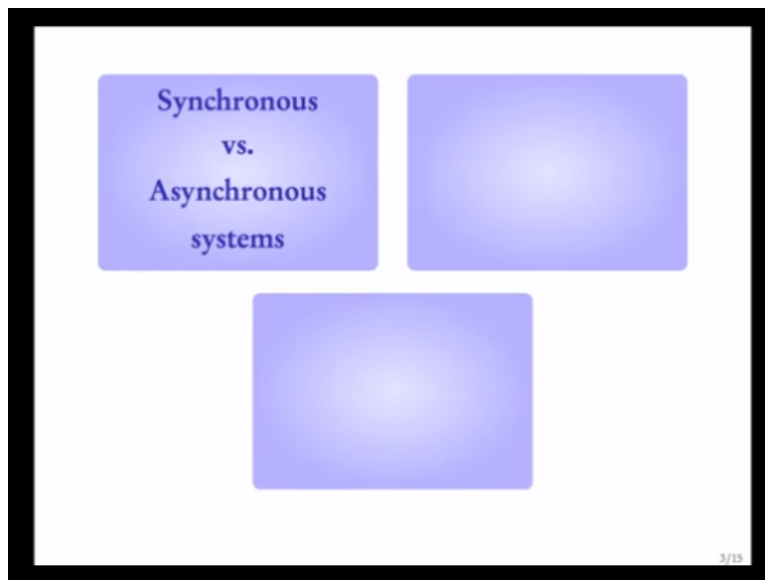
Model Checking
Prof. B. Srivathsan
Department of Computer Science and Engineering
Indian Institute of Technology – Madras

Lecture - 10

Modeling Concurrent Systems in NuSMV

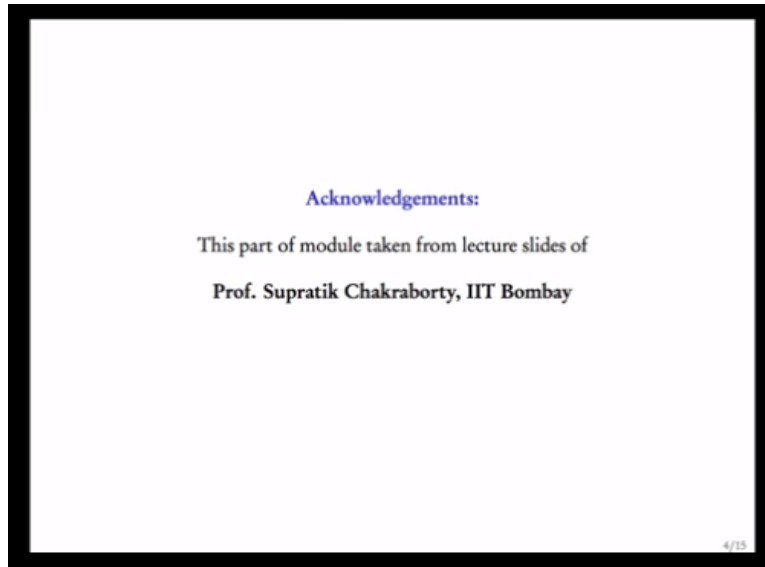
In module 3 we gave examples of modeling hardware circuits using NuSMV. In this module we will focus on different kinds of parallel systems in other words concurrent systems.

(Refer Slide Time: 00:17)



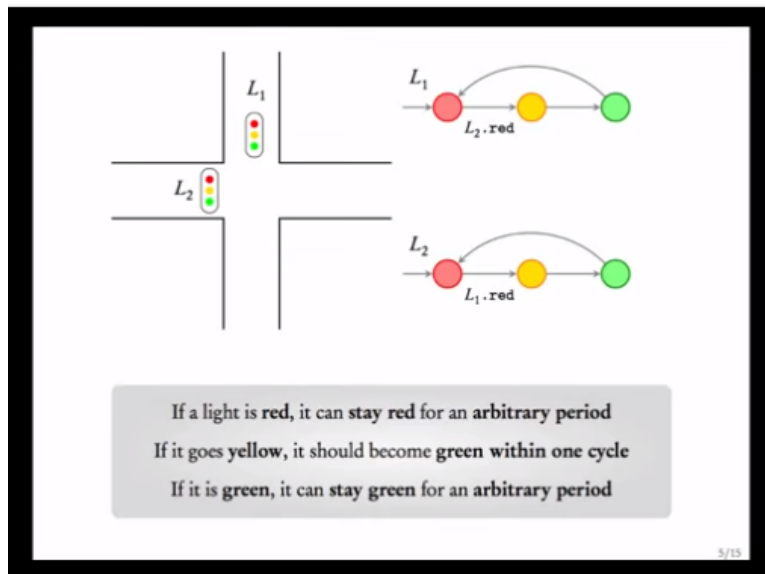
We will start by describing by 2 kinds of parallel systems synchronous and asynchronous. We will understand each of them through examples.

(Refer Slide Time: 00:32)



The content in this part of the module is taken from the lecture slides of Professor. Supratik Chakraborty, IIT, Bombay.

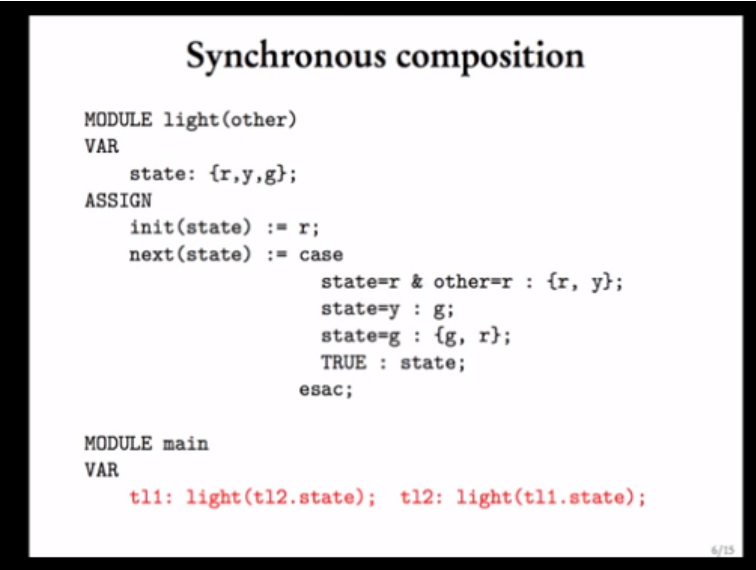
(Refer Slide Time: 00:38)



Here is an example of a road intersection with 2 traffic light signals, light l_1 and l_2 . Light l_1 is described by the transition system here, it has 3 states red, yellow, green. Similarly, the traffic light l_2 is represented by this transition system with the same set of states red, yellow, green. However, they are interacting light l_1 if it is in red can go to yellow only if l_2 is currently in its red state.

Similarly, l2 if it is red it can go to yellow only if l1 is in its red state. Here are some other specifications if a light is red it can stay red for an arbitrary period, if the light goes yellow it should become green within one cycle, what is that mean? the next transition that l1 takes needs to be green, if it is green then the light can stay green for an arbitrary period.

(Refer Slide Time: 02:03)



```

Synchronous composition

MODULE light(other)
VAR
    state: {r,y,g};
ASSIGN
    init(state) := r;
    next(state) := case
        state=r & other=r : {r, y};
        state=y : g;
        state=g : {g, r};
        TRUE : state;
    esac;

MODULE main
VAR
    t1: light(t12.state); t12: light(t11.state);

```

6/15

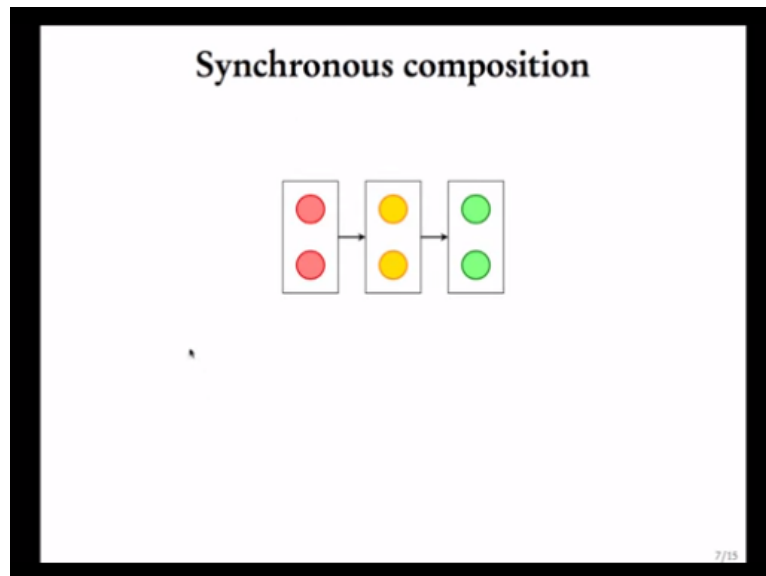
Let us now try to write NuSMV code for this system of traffic lights. We define a module light which takes as input the state of the other light. Module light has variables state, state can take 3 values red, yellow, green, denoted by r, y, g; the initial value of state is r. Lets us now define the transitions. The next value of state is given as follows if the current value of state is r and the state of other light is r as well then the light can either stay in red or it can move to yellow.

This r is to denote the fact that it can stay red for an arbitrary period since other is r it can also go to yellow. If the current state is yellow then the next transition that it takes should make it green. If the state is green it can either stay green or it can go to red. In all other cases there is no change in the state. This is the module light, how do we use this module in module main? We define 2 variables t1 and t2 of type light; t1 is instantiated with t2 dot state, t2 is instantiated with t1 dot state.

This is the same as the NAND example which we saw in the last module. So, what work the states of the transition system defined by this code d? We need to look at the variables in module main, there are 2 variables tl1 and tl2, tl1 is of type light which has a variable state, tl2 is again a variable of type light which has a variable state. So a state of this combined transition will consist of the state of traffic light 1 and the state of traffic light 2. Each time we take the next transition both tl1 and tl2 would move.

That means this is a synchronous composition in every unit both tl1 and tl2 changed their state.

(Refer Slide Time: 04:49)

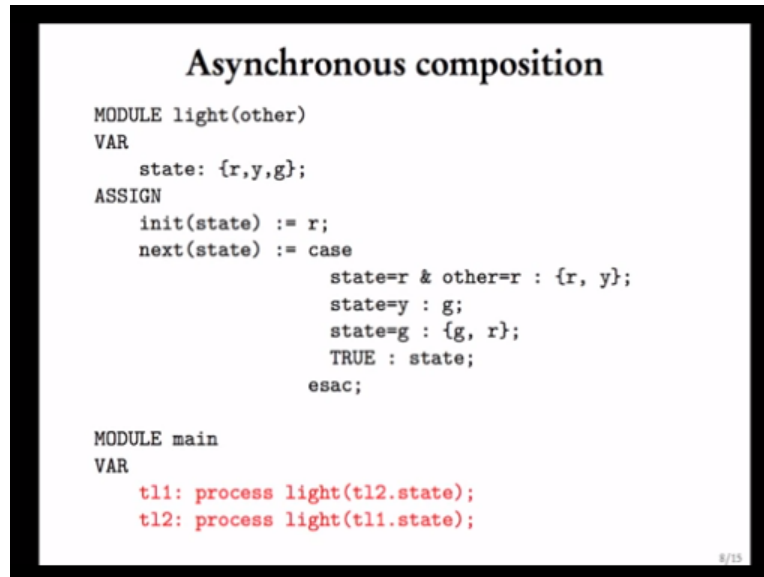


In such a case what will happen? Let us see, we start with tl1 dot state and tl2 dot state being red. In the next unit what are the possible transitions? Since tl2 dot red is true tl1 can indeed to take the transition to yellow. Similarly, since tl1 is red tl2 can indeed satisfy this condition and take the transition to yellow.

We reach a state where both the lights are yellow. Once they are in this state in the next transition both of them can become green. This is a scenario which we did not want this is making both the lights become green simultaneously.

How do we get rid of this behavior? This does not model the system that we want. In our system each of the lights move separately, at a time only one of the light should take the transition. How can we model such systems using NuSMV?

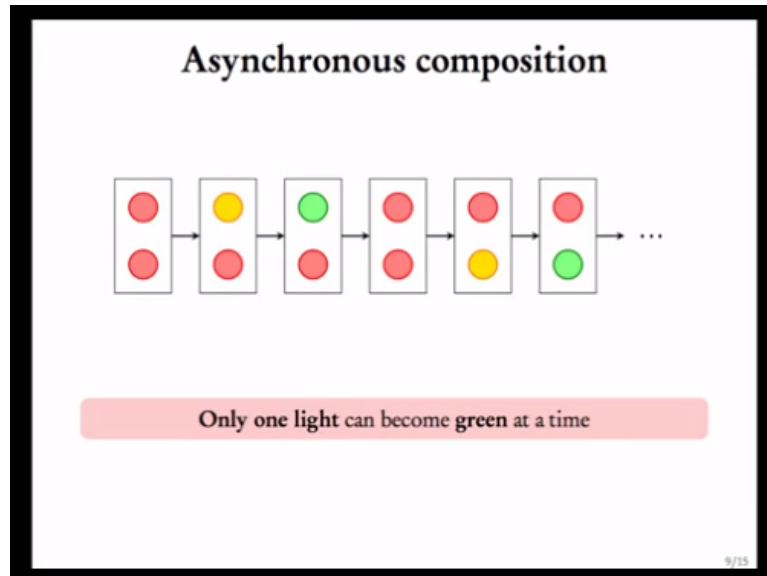
(Refer Slide Time: 06:09)



Firstly, such systems are set to be asynchronous they are not it just means that they are not synchronous. That means in every unit it is not necessary that both of them should simultaneously take the next transition. In the next unit either light 1 can move or light 2 can move. To model such scenario we will define `t1` to be a process light of `t12` dot state. This is the same however we define it with a keyword `process`.

Similarly, `t2` is defined as process light of `t11` dot state what happens when we define it this way.

(Refer Slide Time: 06:54)



The initial state is going to be the same both of them are in red. In the next step one of the lights is chosen to take its next transition. Here light 1 is chosen to take the next transition since light 2 is red, light 1 can indeed move to yellow. In this state yet again one of the lights is chosen to take the next transition.

Suppose, we choose this it can go to green and the other light stays in the same state. Again, we can choose one of the lights to take the next transition here again we choose light 1 it moves to red. Once again we are in the state where both of them are red. Now the next step could be a choice between light 1 and light 2 here we have described the situation where light 2 takes the next transition since light 1 is red it can move to yellow.

Again, here we choose light 2 to take the next transition it becomes green and so on. In a such case only one light can become green at a time. Let us try to run this example using our tool.

(Refer Slide Time: 08:16)

```

GNU nano 2.0.6                                File: light-asyn-demo.smv
MODULE light(other_state)
VAR
    state: {r, y, g};
ASSIGN
    init(state) := r;
    next(state) := case
        state=r & other_state=r: {r,y};
        state=y : g;
        state=g : {g,r};
        TRUE: state;
        esac;

MODULE main
VAR
    tl1: process light(tl2.state);
    tl2: process light(tl1.state);

```

Here is the code, see we have written tl1 to be processed light of tl2 dot state, tl2 is process light tl1 dot state.

(Refer Slide Time: 08:29)

```

srivathsan:Examples sri$ nano light-asyn-demo.smv
srivathsan:Examples sri$ NuSMV -int light-asyn-demo.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 >

```

Let me run this example, when you run it you will get warnings which say that processes are still supported but deprecated in the future processes may no longer be supported. However, we will use this version of NuSMV the goal is to just understand the idea behind a synchronous composition and the use of processes. If not NuSMV there are other tools which support a synchronous composition like SPIN.

The ideas are going to be same, so as long as you understand this you can use any tool that can support a synchronous composition. Let us come back to this example, let's check if all the executions satisfy the condition that both the lights are not green at the same time. How do we check this?

If you remember we use the requirement f in the last module we will use that again. We want to check that in every execution it is not the case that there is a state where both the lights are green. NuSMV says that the specification is indeed true. Let us now try to simulate and understand what happens inside the transition system. For start what are the reachable states.

(Refer Slide Time: 10:21)

```
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 "! F (tl1.state=g & tl2.state=g)"
-- specification !( F (tl1.state = g & tl2.state = g)) is true
NuSMV > print_reachable_states -v
#####
system diameter: 3
reachable states: 5 (2^2.32193) out of 9 (2^3.16993)
----- State 1 -----
tl1.state = r
tl2.state = g
----- State 2 -----
tl1.state = r
tl2.state = r
----- State 3 -----
tl1.state = r
tl2.state = y
----- State 4 -----
tl1.state = g
tl2.state = r
----- State 5 -----
tl1.state = y
tl2.state = r
-----
#####
NuSMV > ||
```

There are 8 sorry there are 4 states given by traffic light 1 is red, 2 is green, both are red, first 1 is red and other is yellow, first one is green and other red, first one is yellow and the other is red. So, there is no state where both are green. Let me now pick an initial state there is only one initial state where both are red.

(Refer Slide Time: 10:47)


```

system diameter: 3
reachable states: 5 (2^2.32193) out of 9 (2^3.16993)
----- State 1 -----
  tl1.state = r
  tl2.state = g
----- State 2 -----
  tl1.state = r
  tl2.state = r
----- State 3 -----
  tl1.state = r
  tl2.state = y
----- State 4 -----
  tl1.state = g
  tl2.state = r
----- State 5 -----
  tl1.state = y
  tl2.state = r
-----
#####
NuSMV > pick_state -i

***** AVAILABLE STATES *****

===== State =====
0) -----
  tl1.state = r
  tl2.state = r

There's only one available state. Press Return to Proceed.

```

Lets us now simulate for some steps to understand what happens inside. We started with both of them being red. Now, let me explain the available states there are actually 3 processes tl1 is a process, tl2 is a process and the main module is also considered to be a process so main dot running is just written as running. If from this state where both are red the selected process is tl2, then tl2 state changes see tl2 dot state becomes yellow.

(Refer Slide Time: 11:40)

```

Chosen state is: 0
NuSMV > simulate -i -k 3
***** Simulation Starting From State 1.1 *****

***** AVAILABLE STATES *****

===== State =====
  tl1.state = r
  tl2.state = y

This state is reachable through:
0) -----
  _process_selector_ = tl2
  running = FALSE
  tl2.running = TRUE
  tl1.running = FALSE

===== State =====
  tl2.state = r

This state is reachable through:
1) -----
  _process_selector_ = tl1
  running = FALSE
  tl2.running = FALSE
  tl1.running = TRUE

2) -----

```

Now, what is this state? rather let us see, if tl1 is the process then we know that process 1 can still stay in red. So that is what it has chosen this state is given by tl2 dot state equal to r and tl1 dot state equal to r, process 1 has been selected and process 1 has chosen to stay red.

(Refer Slide Time: 12:07)

```
tl2.state = y

This state is reachable through:
0) -----
   _process_selector_ = tl2
   running = FALSE
   tl2.running = TRUE
   tl1.running = FALSE

===== State =====
tl2.state = r

This state is reachable through:
1) -----
   _process_selector_ = tl1
   running = FALSE
   tl2.running = FALSE
   tl1.running = TRUE

2) -----
   _process_selector_ = main
   running = TRUE
   tl1.running = FALSE

3) -----
   _process_selector_ = tl2
   running = FALSE
   tl2.running = TRUE
```

There is another way of getting the same process where the process selected is main in which case, both tl1 and tl2 do not change their states. In this example whenever main is selected there would not be any change in the state. However, there could be examples where main has other variables and those variables can change their values.

(Refer Slide Time: 12:31)

```
0) -----
   _process_selector_ = tl2
   running = FALSE
   tl2.running = TRUE
   tl1.running = FALSE

===== State =====
tl2.state = r
i
This state is reachable through:
1) -----
   _process_selector_ = tl1
   running = FALSE
   tl2.running = FALSE
   tl1.running = TRUE

2) -----
   _process_selector_ = main
   running = TRUE
   tl1.running = FALSE

3) -----
   _process_selector_ = tl2
   running = FALSE
   tl2.running = TRUE

===== State =====
tl1.state = y
```

The other way of getting the state r r is when tl2 is selected but tl2 choose to stay red, i hope this is clear.

(Refer Slide Time: 12:44)

```

===== State =====
t12.state = r

This state is reachable through:
1) -----
   _process_selector_ = t11
   running = FALSE
   t12.running = FALSE
   t11.running = TRUE

2) -----
   _process_selector_ = main
   running = TRUE
   t11.running = FALSE

3) -----
   _process_selector_ = t12
   running = FALSE
   t12.running = TRUE

===== State =====
t11.state = y

This state is reachable through:
4) -----
   _process_selector_ = t11
   running = FALSE
   t12.running = FALSE
   t11.running = TRUE

```

Now what is other state from r r the other possible state is when t11 dot state is y and t12 dot state is r. Since it is not written here you have to see the previous state that is what it means, t12 dot state is r. This is obtained when t11 is chosen and it chooses to go to the yellow state.

(Refer Slide Time: 13:14)

```

3) -----
   _process_selector_ = t12
   running = FALSE
   t12.running = TRUE

===== State =====
t11.state = y

This state is reachable through:
4) -----
   _process_selector_ = t11
   running = FALSE
   t12.running = FALSE
   t11.running = TRUE

Choose a state from the above (0-4): 4

Chosen state is: 4

***** AVAILABLE STATES *****

===== State =====
t11.state = g
t12.state = r

This state is reachable through:
0) -----
   _process_selector_ = t11

```

Let me choose this state, now from t11 dot state equal to y and t12 dot state equal to r from y r what are the possible successors? If t11 is chosen it cannot stay in yellow it will become green that is this state.

(Refer Slide Time: 13:33)

```

t11.running = TRUE

Choose a state from the above (0-4): 4
Chosen state is: 4

***** AVAILABLE STATES *****

===== State =====
t11.state = g
t12.state = r

This state is reachable through:
0) -----
   _process_selector_ = t11
   running = FALSE
   t12.running = FALSE
   t11.running = TRUE

===== State =====
t11.state = y

This state is reachable through:
1) -----
   _process_selector_ = t12
   running = FALSE
   t12.running = TRUE
   t11.running = FALSE

```

The other choice is if t12 is selected since t11 is yellow t12 dot state will remain in red. If main is selected there is no change in the state, so we get back to t11 dot state equal to y and t12 dot state equal to r.

(Refer Slide Time: 13:57)

```

===== State =====
t11.state = g
t12.state = r

This state is reachable through:
0) -----
   _process_selector_ = t11
   running = FALSE
   t12.running = FALSE
   t11.running = TRUE

===== State =====
t11.state = y

This state is reachable through:
1) -----
   _process_selector_ = t12
   running = FALSE
   t12.running = TRUE
   t11.running = FALSE

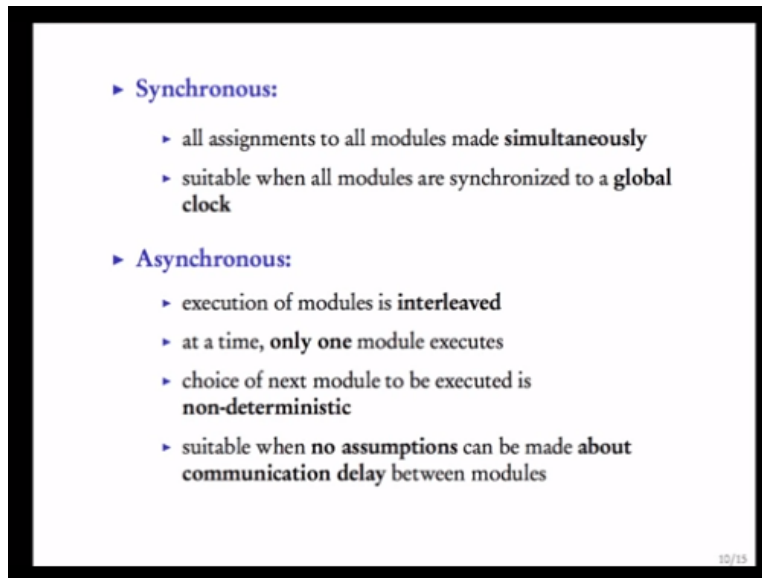
2) -----
   _process_selector_ = main
   running = TRUE
   t12.running = FALSE

Choose a state from the above (0-2): 1

```

You can continue this and understand how the successors transitions are determined. Each time one of the processes is selected either main t12 or t11.

(Refer Slide Time: 14:16)



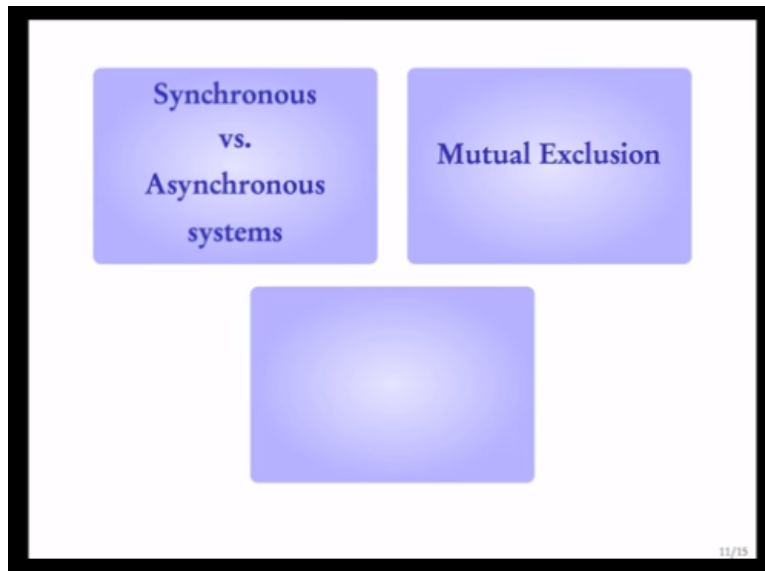
Let us summarize this idea of synchronous versus asynchronous systems. Synchronous modules are defined this way. You have a variable and just instantiate it using the module that you want. For Asynchronous composition you need to have the keyword process before the instantiation.

When the modules are composed in a synchronous way all assignments to all modules are made simultaneously in the next step. Such the composition suitable when all the modules are synchronized to a global clock. The case of the counter and hardware circuit which we saw in the previous module fits well with Synchronous composition.

In the case of Asynchronous composition, the execution of modules is interleaved at a time only one module executes and the choice of the next module to be executed is nondeterministic. This kind of a composition is suitable when no assumptions can be made about the communication delay between modules.

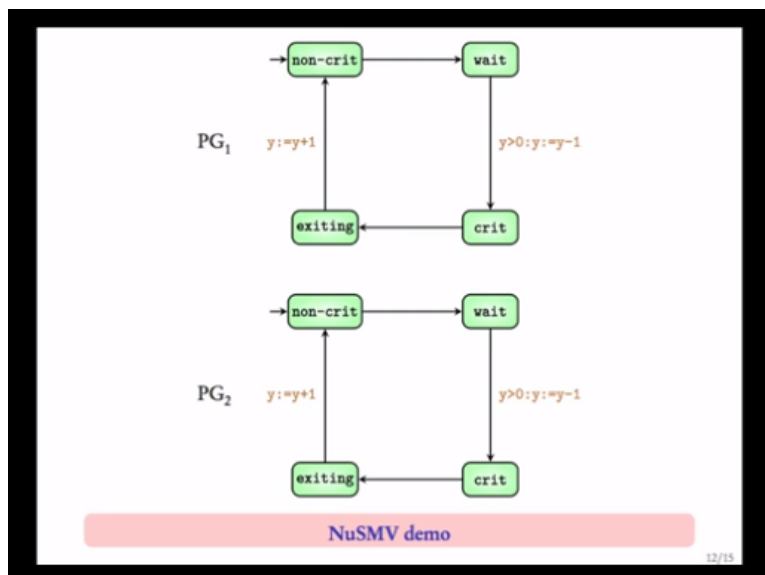
There is no single global clock each of them works with its own clock and we cannot make assumptions about the communication delay between the modules. This brings us to the end of the first part.

(Refer Slide Time: 15:44)



We will now consider Mutual Exclusion. Recall that when 2 programs having a shared resource are running in parallel. The mutual exclusion rule demands that they cannot access the shared resource simultaneously. Sections of the program where the shared resource is access are called critical sections. Mutual exclusion in other words demands that the 2 programs cannot be in their critical section simultaneously.

(Refer Slide Time: 16:29)



Here is a model of 2 parallel programs. To ensure mutual exclusion there is an extra global variable y which can be either 0 or 1. Each program can be in 4 states, its starts with a noncritical state when it wants to access a critical section it goes into a wait state.

In the wait state it can enter the critical section if the value of this global variable y is bigger than 0.

When it enters it decrements the value of y by 1. It can stay in its critical section as long as it wants and when its about to leave it goes into an exiting state. From the exiting state it goes back to the noncritical state in the process it increases the value of y by 1. Program 2 is identical. Let us now try to write the code for these programs in NuSMV and check if they can be in their critical sections simultaneously.

(Refer Slide Time: 18:11)

```
MODULE thread(y)
VAR
  location: {nc, w, c, exit};
ASSIGN
  init(location) := nc;
  next(location) := case
    location=nc : {nc, w};
    location=w & y>0: c;
    location=c : {c, exit};
    location=exit : nc;
  esac;

  next(y) := case
    location=w & y>0: y-1;
    location=exit: y+1;
  esac;

MODULE main
VAR
  y-main: 0..1;
  prg1: process thread(y-main);
  prg2: process thread(y-main);
ASSIGN
  init(y-main):= 1;
```

Let we write the NuSMV code from scratch. Let us define a module called thread which takes as input a variable y . It has 3 locations so we define a variable location it can take 3 values either noncritical sorry it can take 4 values noncritical, waiting, exiting and critical. What are the assignments? The initial value of location is noncritical the next of location is determined as follows.

If location is noncritical the next location can either be noncritical or it can go to the waiting state. If location is wait and the value of y is bigger than 0 then it can go to the critical state. If location is critical then it can either stay in critical or it can go to the exit state and in the exit if location is exit it goes to the noncritical state. What about y ? Next of y is determined as follows.

If location is wait then the value of y is decremented to y minus 1. So location is wait and y is bigger than 0 then you make it y minus 1. If location is exit then we make it y plus 1. How do we use this module thread in module main? we define the following variables. We need to define a variable y let us give it a separate name lets call its a ymain it can take the values 0 and 1.

There are 2 threads so program 1 is defined as process thread of ymain, program 2 is again defined as process thread of ymain. We need to assign the initial value of ymain is true which is 1.

(Refer Slide Time: 22:12)

```
srivathsan:Examples sri$ NuSMV -int mutex-demo1.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. ***

file mutex-demo1.smv: line 13: case conditions are not exhaustive

NuSMV > █
```

Let us now execute this code in NuSMV. NuSMV minus int the name of the file and then we say go, ignore the warnings but there seems to be an error. It says that case conditions are not exhausted.

(Refer Slide Time: 22:35)


```

MODULE thread(y)

VAR
  location: {nc, w, c, exit};

ASSIGN
  init(location) := nc;
  next(location) := case
    location=nc : {nc, w};
    location=w & y>0: c;
    location=c : {c, exit};
    location=exit : nc;
    TRUE: location;
  esac;

  next(y) := case
    location=w & y>0: y - 1;
    location=exit: y+1;
    TRUE: y;
  esac;

MODULE main

VAR
  y-main: 0..1;
  prg1: process thread(y-main);
  prg2: process thread(y-main);

ASSIGN

```

Yes, so we need to write if all these are not satisfied stay in the same location. Similarly, if all these are not satisfied do not change the value of y.

(Refer Slide Time: 22:50)

```

srivathsan:Examples sri$ NuSMV -int mutex-demo1.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. ***

file mutex-demo1.smv: line 16: cannot assign value 2 to variable y-main
NuSMV >

```

Let us now run the new code there seems to be another error it says that cannot assign value 2 to variable ymain.

(Refer Slide Time: 23:09)

```

MODULE thread(y)

VAR
  location: {nc, w, c, exit};

ASSIGN
  init(location) := nc;
  next(location) := case
    location=nc : {nc, w};
    location=w & y>0: c;
    location=c : {c, exit};
    location=exit : nc;
    TRUE: location;
  esac;

  next(y) := case
    location=w & y>0: y - 1;
    location=exit & y=0: y+1;
    TRUE: y;
  esac;

MODULE main

VAR
  y-main: 0..1;
  prg1: process thread(y-main);
  prg2: process thread(y-main);

ASSIGN

```

We need to check this condition and y is strictly less than rather y equal to 0 then make it y plus 1.

(Refer Slide Time: 23:23)

```

srivathsan:Examples sri$ NuSMV -int mutex-demo1.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 >

```

Let us now run it yes, we are successful.

(Refer Slide Time: 24:02)

```

srivathsan:Examples sri$ NuSMV -int mutex-demo1.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 "! F(prg1.location=c & prg2.location=c)"
-- specification !( F (prg1.location = c & prg2.location = c)) is true
NuSMV > █

```

Let us now check the requirement that in all executions program 1 and program 2 cannot be in their critical sections simultaneously. In all executions it is not the case that there exist a state where program 1 dot location is c and program 2 dot location is c and the specification is true. Let us now simulated for a certain number of steps to understand what happens inside.

(Refer Slide Time: 24:15)

```

*** 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 "! F(prg1.location=c & prg2.location=c)"
-- specification !( F (prg1.location = c & prg2.location = c)) is true
NuSMV > pick_state -i

***** AVAILABLE STATES *****

===== State =====
0) -----
   y-main = 1
   prg1.location = nc
   prg2.location = nc

There's only one available state. Press Return to Proceed. █

```

The initial state is when ymain is 1, program 1 is a noncritical location, program 2 is a noncritical location.

(Refer Slide Time: 24:34)

```
There's only one available state. Press Return to Proceed.
```

```
Chosen state is: 0
```

```
NuSMV > simulate -i -k 3
```

```
***** Simulation Starting From State 1.1 *****
```

```
***** AVAILABLE STATES *****
```

```
===== State =====
```

```
y-main = 1  
prg1.location = nc  
prg2.location = w
```

```
This state is reachable through:
```

```
0) -----  
_process_selector_ = prg2  
running = FALSE  
prg2.running = TRUE  
prg1.running = FALSE
```

```
===== State =====
```

```
prg2.location = nc
```

```
This state is reachable through:
```

```
1) -----  
_process_selector_ = prg1  
running = FALSE
```

From the first location from the first state it can go to a state where program 2 has gone to waiting if process 2 rather program 2 is selected to move forward. It can go to a state where both of them are noncritical. This is possible if the process selected is program 1 or program 2 and they choose to stay in the same state. The other possibility is if the main process is selected then we will still get to the same state.

(Refer Slide Time: 25:04)

```
This state is reachable through:
```

```
0) -----  
_process_selector_ = prg2  
running = FALSE  
prg2.running = TRUE  
prg1.running = FALSE
```

```
===== State =====
```

```
prg2.location = nc
```

```
This state is reachable through:
```

```
1) -----  
_process_selector_ = prg1  
running = FALSE  
prg2.running = FALSE  
prg1.running = TRUE
```

```
2) -----  
_process_selector_ = main  
running = TRUE  
prg1.running = FALSE
```

```
3) -----  
_process_selector_ = prg2  
running = FALSE  
prg2.running = TRUE
```

I mean we have essentially not change state both program 1 dot location is nc and program 2 dot location is nc.

(Refer Slide Time: 25:20)

```

===== State =====
y-main = 1
prg1.location = w
prg2.location = w

This state is reachable through:
1) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE

===== State =====
prg2.location = nc

This state is reachable through:
2) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE

3) -----
   _process_selector_ = main
   running = TRUE
   prg2.running = FALSE

Choose a state from the above (0-3): █

```

If program 1 dot location is w and program 2 dot location is nc this means that the process selected is 1 and it has gone to waiting.

(Refer Slide Time: 26:00)

```

===== State =====
y-main = 1
prg1.location = w
prg2.location = w

This state is reachable through:
1) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE
   |

===== State =====
prg2.location = nc

This state is reachable through:
2) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE

3) -----
   _process_selector_ = main
   running = TRUE
   prg2.running = FALSE

Choose a state from the above (0-3): █

```

Let me choose 4 now from where from the state w for program 1 and nc for program 2 we can move to the following states either the first location goes into the critical section or both of them come to wait or program1 stays in wait and program 2 stays in noncritical. Yes, for each of the states there is an explanation as to which process is selected for the next transition you can keep experimenting with this.

(Refer Slide Time: 26:27)

```

===== State =====
y-main = 1
prg1.location = w
prg2.location = w

This state is reachable through:
1) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE

===== State =====
prg2.location = nc

This state is reachable through:
2) -----
   _process_selector_ = prg2
   running = FALSE
   prg2.running = TRUE
   prg1.running = FALSE

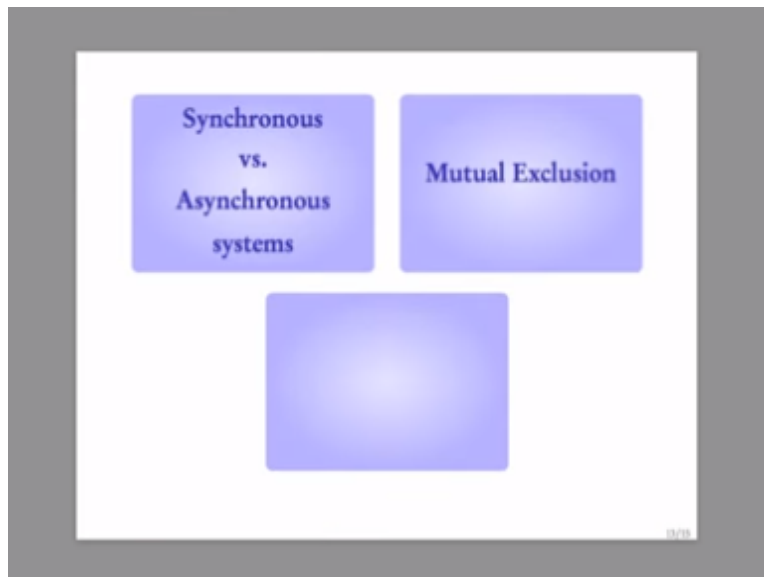
3) -----
   _process_selector_ = main
   running = TRUE
   prg2.running = FALSE

Choose a state from the above (0-3): █

```

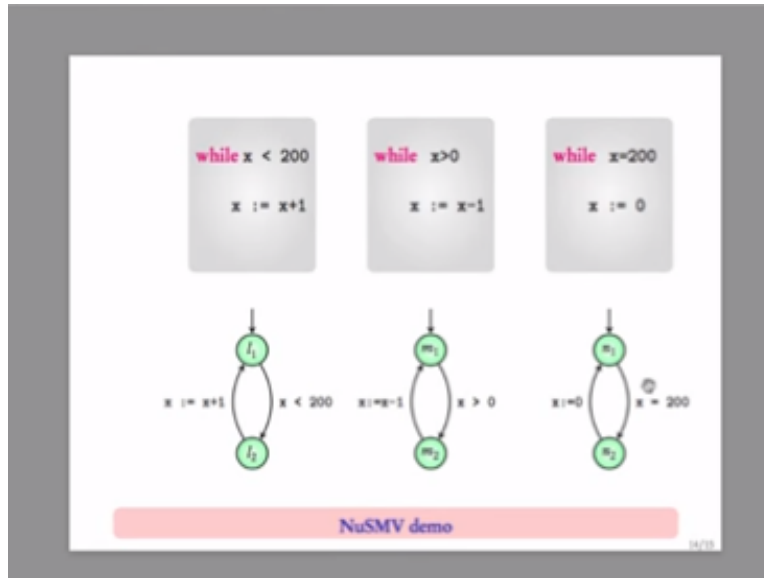
We have therefore seen a demo of a mutual exclusion example.

(Refer Slide Time: 26:38)



In this module we have seen 2 things Synchronous versus Asynchronous systems and a demo of Mutual Exclusion.

(Refer Slide Time: 26:47)



We will now see an example of a system of 3 parallel programs. We had seen this example in module four of unit 1. The first program checks if less than 200 and increments x as long as x is less than 200. This is the transition system corresponding rather this is the program graph corresponding this is the program graph corresponding to this program. 2 locations l1 and l2, l1 to l2 on x less than 200, l2 to l1 on l2 to l1 will set x to x plus 1.

The second program decrements x as long as it is strictly bigger than 0 and the last program checks if x equals 200 and resets x to 0. We want to know if the value of x stays positive always. Let us now write the NuSMV code for this system.

(Refer Slide Time: 31:16)

```

MODULE program1(x1)
VAR
  location : {l1, l2};
ASSIGN
  init(location) := l1;
  next(location) := case
    (location = l1) & x1 < 200 : l2;
    (location = l2) : l1;
    TRUE : location;
  esac;
  next(x1) := case
    (location=l2) : x1 + 1;
    TRUE: x1;
  esac;
MODULE program2(x2)
VAR
  location: {m1, m2};
ASSIGN
  init(location) := m1;
  next(location) := case
    (location = m1) & x2 > 0: m2;
    (location = m2) : m1;
    TRUE : location;
  
```

The first program takes as input an integer x ; we will call it x_1 . It has 2 locations so we define a variable with name `location` it can take values `l1`, `l2` and what are the assignments from `l1` it can go to `l2` and if x is less than 200, from `l2` it can go to `l1`. In the process increments x to $x + 1$ let's type that `init` of `location` is `l1`, `next` of `location` is defined as follows.

If `location` is `l1` and x_1 is less than 200 then go to `l2`, if `location` is `l2` then the next value of `location` changes to `l1`. In all other cases do not change the location. What about the value of the integer? The next of x_1 is determined as follows. If `location` is `l2` then the value of x_1 is incremented by 1 in all other cases do not change its value. Let's now write the second program, it takes again as input one variable let's call it x_2 here.

However we will finally call all of them using the same variable x we will see that later. Again there are 2 locations `m1`, `m2`, How do the transitions look like? From `m1` to `m2` it goes on x greater than 0, from `m2` to `m1` it resets it decrements x by 1. So, `init` of `location` is `m1`, `next` of `location` is determined as follows. If `location` is `m1` and x_2 is bigger than 0 then the next location is `m2`. If `location` is `m2` then the next location is `m1` in all other cases do not change location.

(Refer Slide Time: 31:46)


```

        (location = l1) & x1 < 200 : l2;
        (location = l2) : l1;
        TRUE : location;
    esac;

    next(x1) := case
        (location=l2) : x1 + 1;
        TRUE: x1;
    esac;

MODULE program2(x2)

VAR
    location: {m1, m2};

ASSIGN
    init(location) := m1;
    next(location) := case
        (location = m1) & x2 > 0: m2;
        (location = m2) : m1;
        TRUE : location;
    esac;

    next(x2) := case
        (location = m2) : x2 - 1;
        TRUE: x2;
    esac;

```

M

U174: Where program: demo1.cmu 208 (37,1) : (Test Sys World) (Test Sys World)

What about next of x2? If location is m2 then decrease x2 by 1 in all other cases do not change x2. Now for the final program it has 2 locations n1, n2, What are the transitions? If x is equal to 200 you can go from n1 to n2 in the transition n2 to n1 x is set to 0.

(Refer Slide Time: 34:21)

```

    esac;

MODULE program3(x3)

VAR
    location: {n1,n2};

ASSIGN
    init(location) := n1;
    next(location) := case
        (location = n1) & x3=200: n2;
        TRUE: location;
    esac;

    next(x3) := case
        (location = n2) : 0 ;
        TRUE : x3;
    esac;

MODULE main

VAR
    x: -1000 .. 1000;
    thread1 : process program1(x);
    thread2 : process program2(x);
    thread3 : process program3(x);

```

U174: Where program: demo1.cmu 209 (37,34) : (Test Sys World) (Test Sys World)

Init of location is n1, next of location is if location is n1 and x3 equals 200 then go to n2. In all other cases remain wherever you are. What about next of x3? If location is n2 then make it 0 in all other cases do not change it. Now is the time for the main module, we will define a variable x NuSMV supports only bounded integers.

So, we need to give a bound for x i will give a loose bound saying that my x can range from minus 1000 to 1000. Now we will define 3 instantiations of these programs, thread 1 will be process program 1 of x , thread 2 will be process program 2 of x and thread 3 will be process program 3 of x i have sent the same x to all the programs. However, since x is a bounded integer we need to make a few changes to the conditions in the programs.

(Refer Slide Time: 35:14)

```

        (location = l1) & x1 < 200 : l2;
        (location = l2) : l1;
        TRUE : location;
    esac;
    next(x1) := case
        (location=l2) & x1 < 1000: x1 + 1;
        TRUE: x1;
    esac;

MODULE program2(x2)
VAR
    location: {m1, m2};
ASSIGN
    init(location) := m1;
    next(location) := case
        (location = m1) & x2 > 0: m2;
        (location = m2) : m1;
        TRUE : location;
    esac;

    next(x2) := case
        (location = m2) & x2 > -1000: x2 - 1;
        TRUE: x2;
    esac;

MODULE program3(x3)

```

For example when we are doing this increment we should also check that $x1$ is less than 1000 and when we are doing this decrement we should check if $x2$ is strictly bigger than minus 1000 because we know that x this x is can take values from minus 1000 to plus 1000 only and if they do not give these conditions then NuSMV would complain.

(Refer Slide Time: 35:27)

```

        (location = n1) & x3=200: n2;
        TRUE: location;
      esac;

      next(x3) := case
        (location = n2) : 0 ;
        TRUE : x3;
      esac;

MODULE main
VAR
  x: -1000 .. 1000;
  thread1 : process program1(x);
  thread2 : process program2(x);
  thread3 : process program3(x);

ASSIGN
  init(x) := 0;

```

Let us also assign the initial value of x to be 0. Let us now run the code using NuSMV it takes a while and it is done.

(Refer Slide Time: 36:08)

```

-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  x = 0
  thread1.location = l1
  thread2.location = m1
  thread3.location = n1
-> Input: 1.2 <-
  _process_selector_ = thread1
  running = FALSE
  thread3.running = FALSE
  thread2.running = FALSE
  thread1.running = TRUE
-> State: 1.2 <-
  thread1.location = l2
-> Input: 1.3 <-
-> State: 1.3 <-
  x = 1
  thread1.location = l1
-> Input: 1.4 <-
-> State: 1.4 <-
  thread1.location = l2
-> Input: 1.5 <-
-> State: 1.5 <-
  x = 2
  thread1.location = l1
-> Input: 1.6 <-
-> State: 1.6 <-
  thread1.location = l2

```

Let us now check if the value of x is always bigger than or equal to 0, g x bigger than or equal to 0. It says it is false and it is given us an execution with 407 states. Let us now try to understand what is happening. So, let's get first to the beginning of this execution. Yeah, the counter example starts with the initial state where location of thread 1 is l1, thread 2 is an m1, thread 3 is an n1 and the value of x is 0. Now the value of x is incremented by thread 1 repeatedly for certain amount of steps.

(Refer Slide Time: 36:59)

```

x = 3
thread1.location = l1
-> Input: 1.8 <-
-> State: 1.8 <-
thread1.location = l2
-> Input: 1.9 <-
-> State: 1.9 <-
x = 4
thread1.location = l1
-> Input: 1.10 <-
-> State: 1.10 <-
thread1.location = l2
-> Input: 1.11 <-
-> State: 1.11 <-
x = 5
thread1.location = l1
-> Input: 1.12 <-
-> State: 1.12 <-
thread1.location = l2
-> Input: 1.13 <-
-> State: 1.13 <-
x = 6
thread1.location = l1
-> Input: 1.14 <-
-> State: 1.14 <-
thread1.location = l2
-> Input: 1.15 <-
-> State: 1.15 <-
x = 7
thread1.location = l1

```

So, x is incremented process 1 goes from l1, l2, l1, l2 and so on and x is getting incrementing till x reaches 200 i guess. Let us check it you see it is the same l1, l2 l1, l2, l1, l2, x is becoming 157, 159, x is becoming 198, 199 and now there is a slight change.

(Refer Slide Time: 38:19)

```

thread1.location = l1
-> Input: 1.396 <-
-> State: 1.396 <-
thread1.location = l2
-> Input: 1.397 <-
-> State: 1.397 <-
x = 198
thread1.location = l1
-> Input: 1.398 <-
-> State: 1.398 <-
thread1.location = l2
-> Input: 1.399 <-
-> State: 1.399 <-
x = 199
thread1.location = l1
-> Input: 1.400 <-
-> State: 1.400 <-
thread1.location = l2
-> Input: 1.401 <-
  _process_selector_ = thread2
thread2.running = TRUE
thread1.running = FALSE
-> State: 1.401 <-
thread2.location = m2
-> Input: 1.402 <-
  _process_selector_ = thread1
thread2.running = FALSE
thread1.running = TRUE
-> State: 1.402 <-
x = 200

```

Process 2 is selected and it goes into m2. So, currently the states are l2, m2 and n1. Process 1 is selected it goes to l1 and increments by 200. Right now the state is thread 1 is l1, thread 2 is n2, thread 3 is n1 and the value of x is 200.

(Refer Slide Time: 38:49)

```

thread1.location = l2
-> Input: 1.399 <-
-> State: 1.399 <-
  x = 199
  thread1.location = l1
-> Input: 1.400 <-
-> State: 1.400 <-
  thread1.location = l2
-> Input: 1.401 <-
  _process_selector_ = thread2
  thread2.running = TRUE
  thread1.running = FALSE
-> State: 1.401 <-
  thread2.location = m2
-> Input: 1.402 <-
  _process_selector_ = thread1
  thread2.running = FALSE
  thread1.running = TRUE
-> State: 1.402 <-
  x = 200
  thread1.location = l1
-> Input: 1.403 <-
  _process_selector_ = thread3
  thread3.running = TRUE
  thread1.running = FALSE
-> State: 1.403 <-
  thread3.location = n2
-> Input: 1.404 <-
-> State: 1.404 <-
  x = 0

```

So, now thread 3 can execute because there was a condition from going to go from n1 to n2 it needed the value of x to be 200. So, right now thread 3 is in location n2 it can reset its value to 0 once it's in n2 and then it should change back to n1. So the value of x become 0 however, there is no change to the value of the location of thread 3.

(Refer Slide Time: 39:43)

```

(location = m2) & (x2 > -1000): x2 - 1;
TRUE: x2;
esac;

MODULE program3(x3)
VAR
  location: {n1,n2};
ASSIGN
  init(location) := n1;
  next(location) := case
    (location = n1) & x3=200: n2;
    (location = n2) : n1;
    TRUE: location;
    esac;

  next(x3) := case
    (location = n2) : 0 ;
    TRUE : x3;
    esac;

MODULE main
VAR
  x: -1000 .. 1000;
  thread1 : process program1(x);

```

Normally it should have gone to n1 let us now check the code. Right, we have missed in the code to say that if location is n2 and sorry if location is n2 you need to go to n1. We can run it with this now however there should be no change to the counter example. Let us first understand this counter example, it goes to 0 and then thread 2 which is in n2 decrements it by 1 and hence the value of x becomes minus 1 so this is the situation.

(Refer Slide Time: 40:02)

```
x = 200
thread1.location = l1
-> Input: 1.403 <-
  _process_selector_ = thread3
  thread3.running = TRUE
  thread1.running = FALSE
-> State: 1.403 <-
  thread3.location = n2
-> Input: 1.404 <-
-> State: 1.404 <-
  x = 0
-> Input: 1.405 <-
  _process_selector_ = thread2
  thread3.running = FALSE
  thread2.running = TRUE
-> State: 1.405 <-
  x = -1
  thread2.location = m1
-> Input: 1.406 <-
  _process_selector_ = thread3
  thread3.running = TRUE
  thread2.running = FALSE
-- Loop starts here
-> State: 1.406 <-
  x = 0
-> Input: 1.407 <-
  _process_selector_ = main
  running = TRUE
  thread3.running = FALSE
-- Loop starts here
```

Let us now run the corrected code the requirement $g \ x$ bigger than or equal to 0 should be false even now and we should be getting a similar counter example let us check it. Check $ltl \ spec \ minus \ p \ g \ of \ x \ bigger \ than \ or \ equal \ to \ 0$, its false and we seem to have gotten the same example now with the corrected state as well.

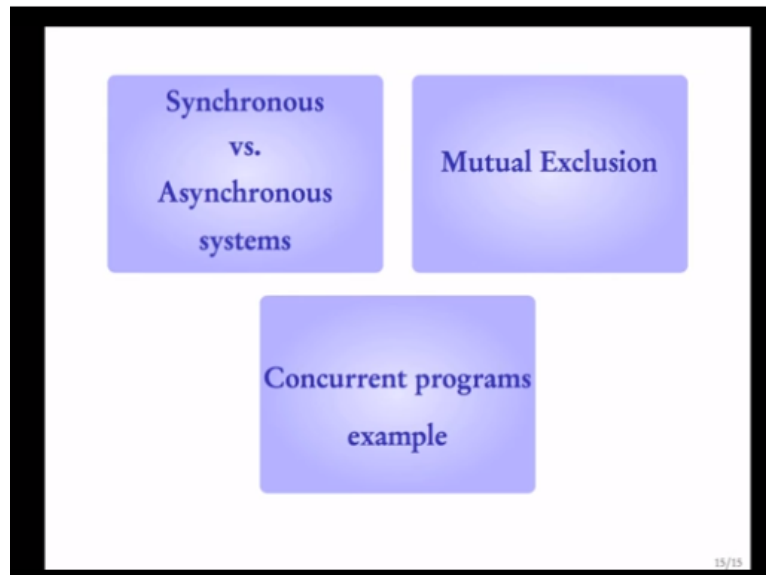
(Refer Slide Time: 40:41)

```
thread1.running = FALSE
-> State: 1.401 <-
  thread2.location = m2
-> Input: 1.402 <-
  _process_selector_ = thread1
  thread2.running = FALSE
  thread1.running = TRUE
-> State: 1.402 <-
  x = 200
  thread1.location = l1
-> Input: 1.403 <-
  _process_selector_ = thread3
  thread3.running = TRUE
  thread1.running = FALSE
-> State: 1.403 <-
  thread3.location = n2
-> Input: 1.404 <-
-> State: 1.404 <-
  x = 0
  thread3.location = n1
-> Input: 1.405 <-
  _process_selector_ = thread2
  thread3.running = FALSE
  thread2.running = TRUE
-> State: 1.405 <-
  x = -1
  thread2.location = m1
-> Input: 1.406 <-
  _process_selector_ = thread1
  thread2.running = FALSE
```

Thread 3 dot location goes from n2 to n1 when x is set to 0. Notice that it is possible to write models erroneously to some extent these simulations will help you identify the errors while writing the models. However, there is some manual responsibility involved

in writing correct models. NuSMV will just ensure that if you give a model and give a requirement it will check if that requirement is true on that model.

(Refer Slide Time: 41:18)



This brings us to the end of this module. We have seen 3 things the first was an example of Synchronous versus Asynchronous systems, secondly we saw an example of Mutual Exclusion. We checked it using NuSMV and finally we saw an example of Concurrent Programs. We have seen how NuSMV can be used to check requirements on models of these systems.