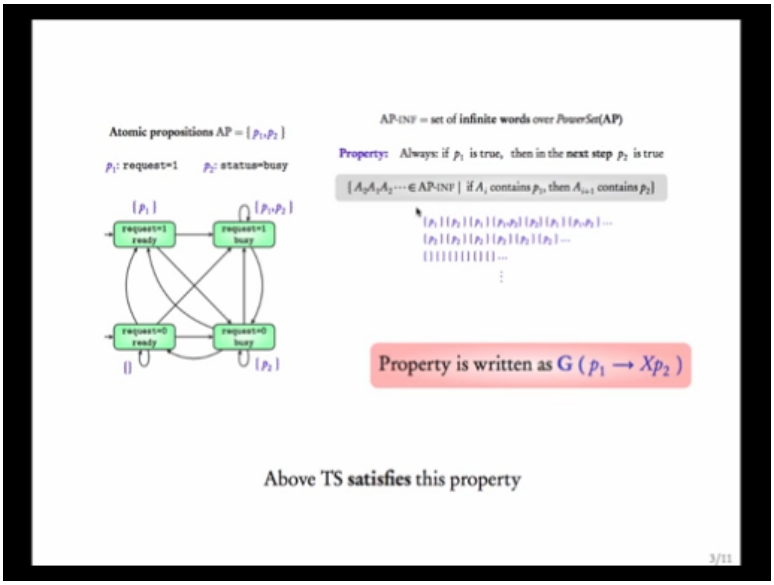
**Model Checking**
**Prof. B. Srivathsan**
**Department of Computer Science and Engineering**
**Indian Institute of Technology – Madras**

**Lecture - 15**
**Safety Properties**

In module 2, we look that properties as a set of words. In module 3 we saw a special kind of property called invariants. Invariants contain words in which every letter satisfies a condition. In this module we will see an extension of invariant properties called safety properties.

**(Refer Slide Time: 00:33)**




We will start with an example, consider this transition system that we are familiar with we want to check this property it says that always if p1 is true then in the next step p2 is true. Lets us look at some words which satisfies this property. Look at this word here p1 is true then in the next step p2 should be true yes indeed, here p1 is true. So, in the next step p2 should be true, here again p1 is true so p2 should be true here, here p1 is true so p2 should be true here and so on.

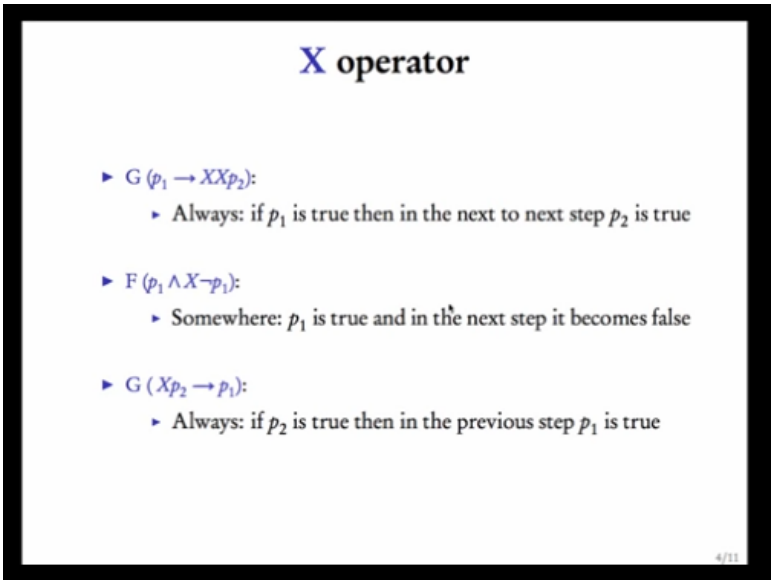
In this word p1 is never true so we have nothing to check same for this word. Since p1 is never true we have nothing to check the condition just says that if p1 is true then in the next step p2 should be true. How do we specify such properties? Let me introduce the x

operator for always we have g if p1 is true then so p1 implies xp2 this will check that if p1 is true then in the next step p2 should be true. It turns out that this transition system satisfies this property that means the traces of this transition system are words of this form.

**(Refer Slide Time: 02:39)**


Let us now check this property in NuSMV. NuSMV request go the command is the same check ltl spec minus p and now we want to check that if request is true implies the next step status is busy. Indeed NuSMV confirms that this specification is true on the transition system.

**(Refer Slide Time: 03:31)**

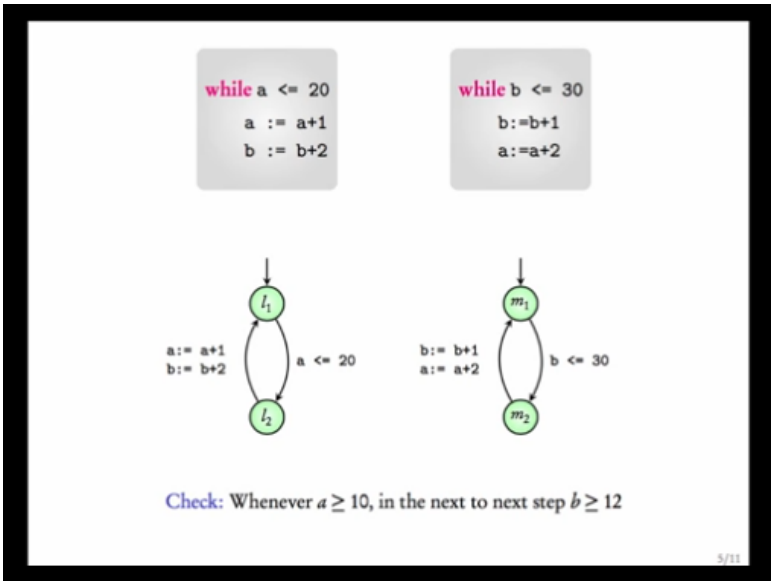


Let us look at some examples of using the x operator this property says that always if p1 is true then p2 is true in the next to next step, the single x goes to one step forward. If you have one more x this means that if p1 is true then the next to next step p2 is true. If you remember f, f says that somewhere p1 and x of not p1 is true that means somewhere in the execution p1 is true and in the next step it becomes false.

So, an execution satisfies this if at some point there is a p1 and in the step p1 is gone. What about this property, this says that g of x of p2 implies p1, this says that if p2 is true than the previous step p1 is true, this says that if next of p2 then p1 that means if p2 is true in the next step p1 should be true now.

In other words if p1 is true then in the previous step p1 sorry if p2 is true then in the previous step p1 should be true. The point of this slide is to give you examples of just using the x operator it is a very simple operator and I hope that the usage of this is clear.

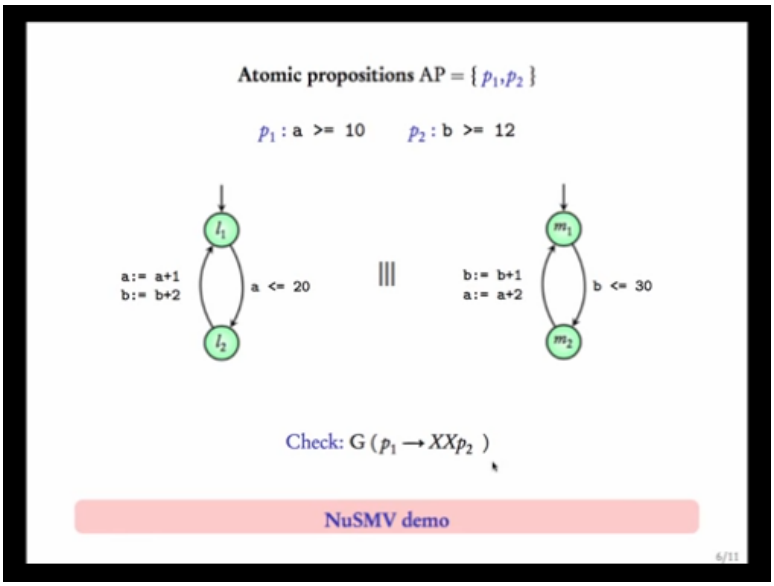
**(Refer Slide Time: 05:16)**




Now, lets us look at another example there are 2 parallel programs communicating programs over variables a and b this checks if is a is less than or equal to 20 it increments a by 1 and b by 2, this one checks if b is less than equal to 30 and increments b by 1 and a by 2 these are the program graphs corresponding to these programs.

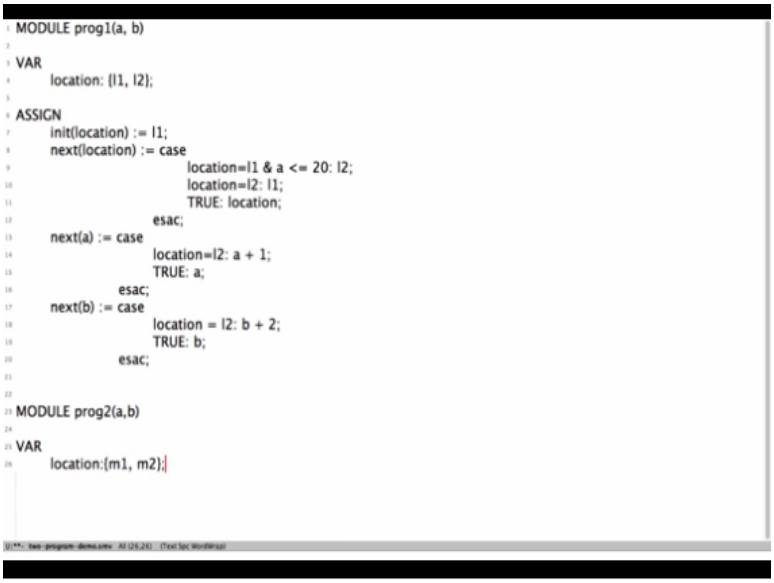
We want to check some properties on the system consisting of these 2 programs we want to check if whenever a is bigger than or equal to 10 in the next to next step b becomes bigger than or equal to 12, this is the property we want to check.

**(Refer Slide Time: 06:15)**




So, first let us define appropriate atomic propositions we will define 2 atomic propositions, the first proposition says that a is bigger than or equal to 10 and the second proposition says that b is bigger than or equal to 12 and the property that we want to check is g f of p1 implies x xp2 and the model is the interleaving of these 2 program graphs. Let us now specify these program graphs in NuSMV and check this condition.

**(Refer Slide Time: 07:06)**



```
MODULE prog1(a, b)

VAR
    location: {l1, l2};

ASSIGN
    init(location) := l1;
    next(location) := case
                        location=l1 & a <= 20: l2;
                        location=l2: l1;
                        TRUE: location;
                    esac;
    next(a) := case
                    location=l2: a + 1;
                    TRUE: a;
                esac;
    next(b) := case
                    location = l2: b + 2;
                    TRUE: b;
                esac;


MODULE prog2(a,b)

VAR
    location:{m1, m2};
```

Let us now define the module for the first program, program 1 it takes as input 2 integers the variables are given by the location l1, l2. Here is the transition function the initial value of location is l1,the next value of location is as follows. If location is l1 and the value of a is less than or equal to 20, then go to l2. If location is l2 then go to l1 in all other cases just stay in the same location. What about next of a, if location is l2 let us look at the program graph.

If location is l2 then a becomes a plus 1 b becomes b plus 2. So, a goes to a plus 1 if otherwise that's all otherwise just keep it a. Now, what about b if location is l2 then it goes to b plus 2 otherwise keep it b. Now, let us write the other program it will be similar it has locations m1, m2 initial value of location is m1. Now what about the transition it is similar lets try to copy this, if location is m1 and b is less than or equal to 30 then go to m2, if location is m2 get back to m1 in other cases just stay.

**(Refer Slide Time: 09:58)**

```
                    location=l1 & a <= 20: l2;
                    location=l2: l1;
                    TRUE: location;
                esac;
    next(a) := case
                    location=l2: a + 1;
                    TRUE: a;
                esac;
    next(b) := case
                    location = l2: b + 2;
                    TRUE: b;
                esac;


MODULE prog2(a,b)

VAR
    location:{m1, m2};

ASSIGN
    init(location) := m1;
    next(location) := case
                    location=m1 & b <= 30: m2;
                    location=m2: m1;
                    TRUE: location;
                esac;
```

How does it modify a and b, let us first copy this from m2 to m1 it increments b by b1 and a by 2.

**(Refer Slide Time: 11:03)**

```
        location:{m1, m2};

ASSIGN
        init(location) := m1;
        next(location) := case
                                location=m1 & b <= 30: m2;
                                location=m2: m1;
                                TRUE: location;
                        esac;
         next(a) := case
                        location=m2 & a <= 98: a + 2;
                        TRUE: a;
                esac;
        next(b) := case
                        location = m2 & b <= 99: b + 1;
                        TRUE: b;
                esac;


MODULE main

VAR
        a: 0 .. 100;
        b: 0 .. 100;
```

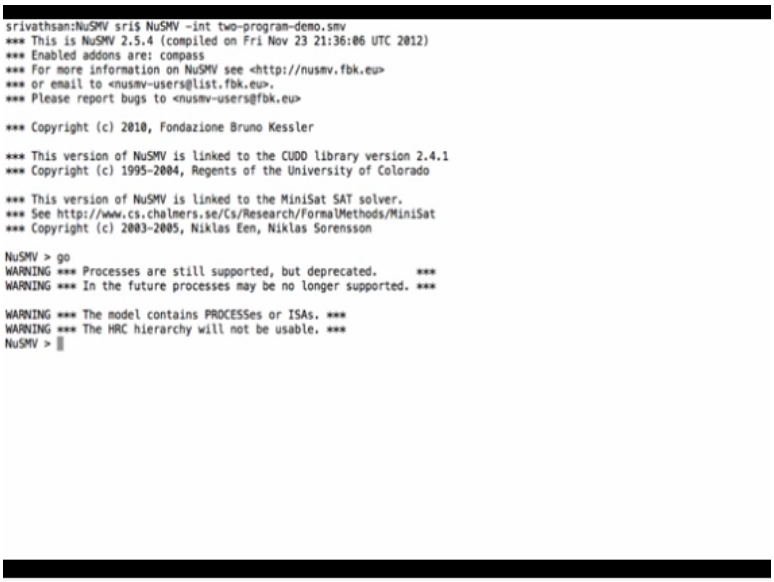
So, if location is m2 increment a by 2, if location is m2 increment b by 1. Now, let us write the main module and define variables a let us give the bound from 0 to 100. Therefore, whenever you do a change we need to also check the boundary condition you can increment it by 2 only if a is less than or equal to 98 here b is less than or equal to 99.

**(Refer Slide Time: 11:15)**

```
MODULE prog1(a, b)

VAR
        location: {l1, l2};

ASSIGN
        init(location) := l1;
        next(location) := case
                                location=l1 & a <= 20: l2;
                                location=l2: l1;
                                TRUE: location;
                        esac;
        next(a) := case
                        location=l2 & a <= 99: a + 1;
                        TRUE: a;
                esac;
        next(b) := case
                        location = l2 & b <= 98: b + 2;
                        TRUE: b;
                esac;


MODULE prog2(a,b)

VAR
        location:{m1, m2};

ASSIGN
        init(location) := m1;
        next(location) := case
```

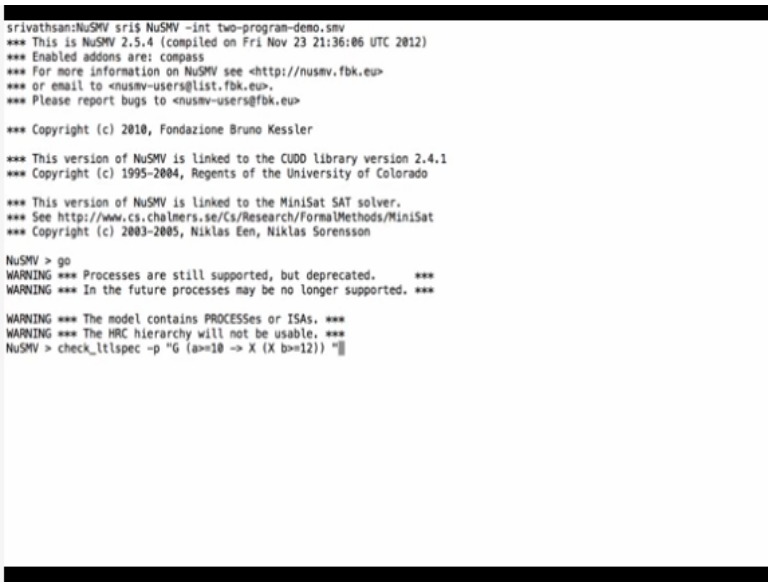
Similarly, here a is less than or equal to 99 and b is less than or equal to 98. We now need to define the processes let us call them pr1 is a process. What was the name program 1 over a comma b, pr2 is process program 2 over a comma b, we just need to assign the initial value of a and the initial value of b. Let us now save this and try to run this program in NuSMV.

**(Refer Slide Time: 12:16)**




So, NuSMV minus int, go there are no errors. Let us now check the property g of p1 implies xxp2 where p1 is a bigger than or equal to 10, p2 is b bigger than or equal to 12.

**(Refer Slide Time: 12:52)**




Check ltlspec minus p, g of a bigger than or equal to 10 implies x, x b bigger than or equal to 12. It says that the property is false and it gives us a counterexample. Let us try to understand this counterexample it starts with both a and b 0 and both of them at l1 and m1, initial state and then process to increments a by 2 and b by 1.

**(Refer Slide Time: 13:17)**

```
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > check_ltlspec -p "G (a>=10 -> X (X b>=12)) "
-- specification  G (a >= 10 ->  X ( X b >= 12))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a = 0
  b = 0
  pr1.location = l1
  pr2.location = m1
-> Input: 1.2 <-
  _process_selector_ = pr2
  running = FALSE
  pr2.running = TRUE
  pr1.running = FALSE
-> State: 1.2 <-
  pr2.location = m2
-> Input: 1.3 <-
-> State: 1.3 <-
  a = 2
  b = 1
  pr2.location = m1
-> Input: 1.4 <-
-> State: 1.4 <-
  pr2.location = m2
-> Input: 1.5 <-
-> State: 1.5 <-
  a = 4
  b = 2
  pr2.location = m1
-> Input: 1.6 <-
-> State: 1.6 <-
  pr2.location = m2
-> Input: 1.7 <-
-> State: 1.7 <-
  a = 6
```

Again process 2 increments a by 2 and b by 1 and so on this keeps going till process 2 becomes bigger than 10 but b is still 5.

**(Refer Slide Time: 13:39)**

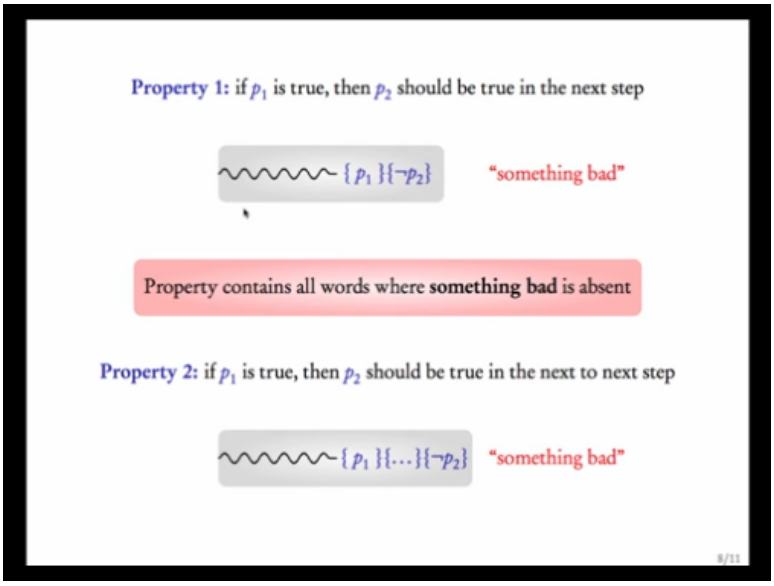
```
WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > check_ltlspec -p "G (a>=10 -> X (X b>=12)) "
-- specification  G (a >= 10 ->  X ( X b >= 12))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a = 0
  b = 0
  pr1.location = l1
  pr2.location = m1
-> Input: 1.2 <-
  _process_selector_ = pr2
  running = FALSE
  pr2.running = TRUE
  pr1.running = FALSE
-> State: 1.2 <-
  pr2.location = m2
-> Input: 1.3 <-
-> State: 1.3 <-
  a = 2
  b = 1
  pr2.location = m1
-> Input: 1.4 <-
-> State: 1.4 <-
  pr2.location = m2
-> Input: 1.5 <-
-> State: 1.5 <-
  a = 4
  b = 2
  pr2.location = m1
-> Input: 1.6 <-
-> State: 1.6 <-
  pr2.location = m2
-> Input: 1.7 <-
```

This shows that the property is not true and here is a trace which exhibits the violation.

**(Refer Slide Time: 13:43)**


We have seen examples of properties using the x operator we will now give you the idea of safety properties.

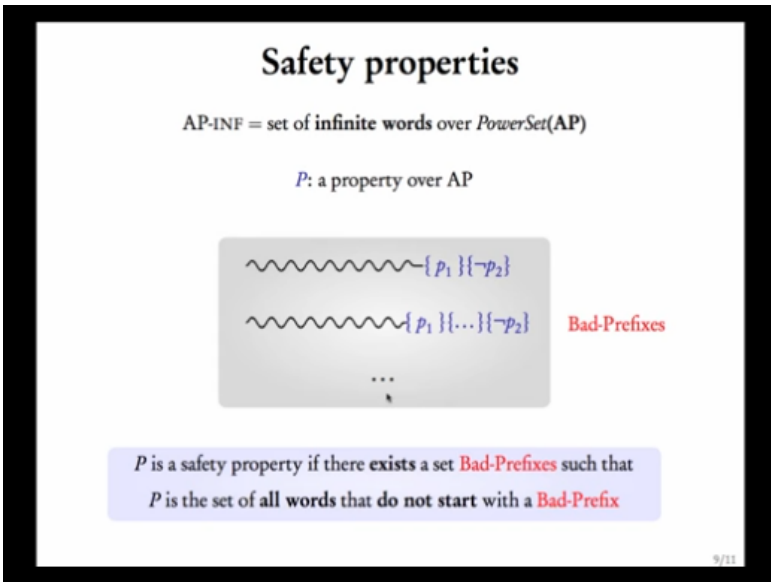
**(Refer Slide Time: 13:57)**




Property 1 was if property 1 is true then p2 should be true in the next step. What is a violation of this property? What is something bad for this property if there is a word which has p1 and in the next step there is no p2 then this situation is something bad for this property. Similarly, look at this property if p1 is true then p2 should be true in the next to next step. What is bad for this property?

Any word like this where there is a p1 followed by something followed by set where p2 is not there this kind of a pattern is something bad for this property. And this property contains all the words where something bad is absent essentially this property contains all the words which do not start with a bad pattern.

**(Refer Slide Time: 15:15)**

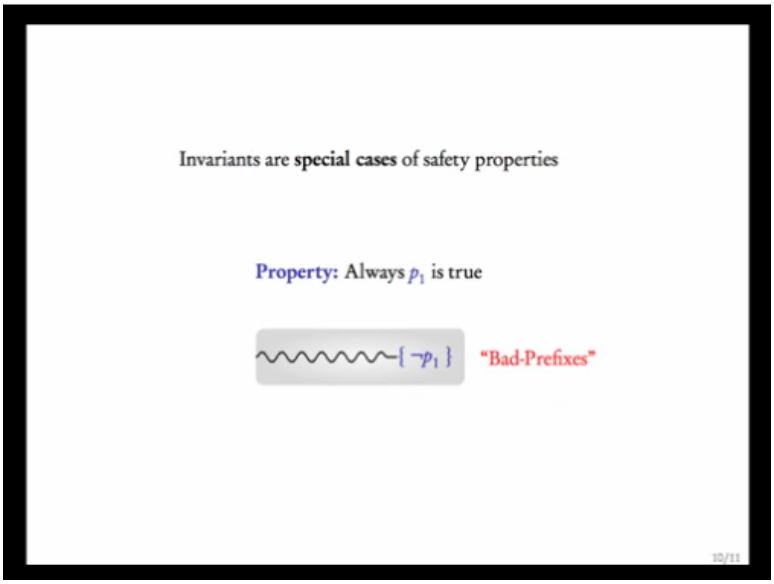



This is the characteristic of safety properties. Let p a property over atomic propositions ap that means it is a set of words over powerset ap, p is a safety property the set of words p is set to be a safety property if there exists a set of bad prefixes such that p contains or other p is exactly the set of all words that do not start with a bad prefix. Let me repeat this again a property is just a set of words, When do you call this property to be a safety property?

That is what I am going to explain now, this property would be called a safety property if you can come up with the set of bad prefixes such that, p is exactly the set of all words which do not start with a bad prefix. For example this was the set of bad prefixes all words where there is a p1 followed by not p2 the property 1 contain all words where this bad prefix is absent. Similarly, property 2 is exactly the set of all words where this bad prefix is absent rather the set of bad prefixes all words where this pattern is present.

This explains the nomenclature safety properties these can be used to verify if something bad never happens.

**(Refer Slide Time: 17:28)**




Note that invariants are special cases of safety properties where the bad prefixes dependent only on one set. The property always p1 is true a bad prefix for it is a word which contains not p1 in the sense that a word where p1 is absent. These are not invariants because here you need to have an idea about the following set. In particular remember that we said invariants can be checked by looking at just the states just the reachable states.

In the case of safety, this is not enough we will see how safety properties are checked later during the course.

**(Refer Slide Time: 18:23)**


In this module you just need to remember that safety properties are properties which are wide bad prefixes they contain all words that avoid some bad prefixes these properties can be used to check that something bad never happens. A very useful operator for such properties is x where x can be used to go to the next step.