

nuXmv model checking

Enrico Magnago

University of Trento,
Fondazione Bruno Kessler

Modelling a Program in nuXmv

Example: model programs in nuXmv [1/4]

Q: given the following piece of code, computing the GCD, how do we *model* and *verify* it with **nuXmv**?

```
void main() {
    ... // initialization of a and b
    while (a!=b) {
        if (a>b)
            a=a-b;
        else
            b=b-a;
    }
    ... // GCD=a=b
}
```

Step 1: label the **entry point** and the **exit point** of every block

```
void main() {  
    ... // initialization of a and b  
11:    while (a!=b) {  
12:        if (a>b)  
13:            a=a-b;  
        else  
14:            b=b-a;  
    }  
15:    ... // GCD=a=b  
}
```

Example: model programs in nuXmv [3/4]

Step 2: encode the transition system with the assign style

```
MODULE main()
VAR  a: 0..100;  b: 0..100;
    pc: {11,12,13,14,15};
ASSIGN
  init(pc) := 11;
  next(pc) :=
    case
      pc=11 & a!=b      : 12;
      pc=11 & a=b       : 15;
      pc=12 & a>b       : 13;
      pc=12 & a<=b     : 14;
      pc=13 | pc=14    : 11;
      pc=15             : 15;
    esac;
```

```
next(a) :=
  case
    pc=13 & a > b: a - b;
    TRUE: a;
  esac;

next(b) :=
  case
    pc=14 & b >= a: b-a;
    TRUE: b;
  esac;
```

Example: model programs in nuXmv [4/4]

Step 2: (alternative): use the constraint style

```
MODULE main
VAR
  a : 0..100;  b : 0..100;  pc : {11, 12, 13, 14, 15};
INIT pc = 11
TRANS
  pc = 11 -> (((a != b & next(pc) = 12) |
              (a = b & next(pc) = 15)) &
             next(a) = a & next(b) = b)
TRANS
  pc = 12 -> (((a > b & next(pc) = 13) |
              (a < b & next(pc) = 14)) &
             next(a) = a & next(b) = b)
TRANS
  pc = 13 -> (next(pc) = 11 & next(a) = (a - b) & next(b) = b)
TRANS
  pc = 14 -> (next(pc) = 11 & next(b) = (b - a) & next(a) = a)
TRANS
  pc = 15 -> (next(pc) = 15 & next(a) = a & next(b) = b)
```

Model Properties

Model Properties [1/2]

A property:

- can be added to any module within a program

```
CTLSPEC AG (req -> AF sum = op1 + op2);
```

- can be specified through NUXMV interactive shell

```
nuXmv > check_ctlspec -p "AG (req -> AF sum = op1 + op2)"
```

Notes:

- `show_property` lists all properties collected in an *internal database*:

```
nuXmv > show_property
**** PROPERTY LIST [ Type, Status, Counter-example Number, Name ]
----- PROPERTY LIST -----
000 :AG !(procl.state = critical & proc2.state = critical)
      [CTL           True           N/A      N/A]
001 :AG (procl.state = entering -> AF procl.state = critical)
      [CTL           True           N/A      N/A]
```

- each property can be verified one at a time using its **database index**:

```
nuXmv > check_ctlspec -n 0
```


Property verification:

- each property is separately verified
- the result is either “TRUE” or “FALSE + counterexample”
 - **Warning:** the generation of a counterexample is not possible for all CTL properties: e.g., temporal operators corresponding to existential path quantifiers cannot be proved false by showing a single execution path

Different kinds of properties are supported:

- **Invariants:** properties on every reachable state
- **LTL:** properties on the computation paths
- **CTL:** properties on the computation tree

- Invariant properties are specified via the keyword `INVARSPEC`:
`INVARSPEC <simple_expression>`
- Invariants are checked via the `check_invar` command

Remark:

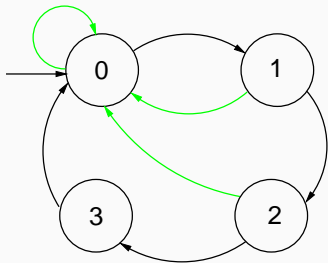
during the checking of invariants, all the fairness conditions associated with the model are ignored

Example: modulo 4 counter with reset [1/2]

```
MODULE main
VAR  b0    : boolean;
     b1    : boolean;
     reset : boolean;
ASSIGN
  init(b0) := FALSE;
  next(b0) := case
    reset : FALSE;
    !reset : !b0;
  esac;
  init(b1) := FALSE;
  next(b1) := case
    reset : FALSE;
    TRUE  : ((!b0 & b1) |
             (b0 & !b1));
  esac;
DEFINE out := toint(b0) + 2*toint(b1);

INVARSPEC out < 2
```

- recall:



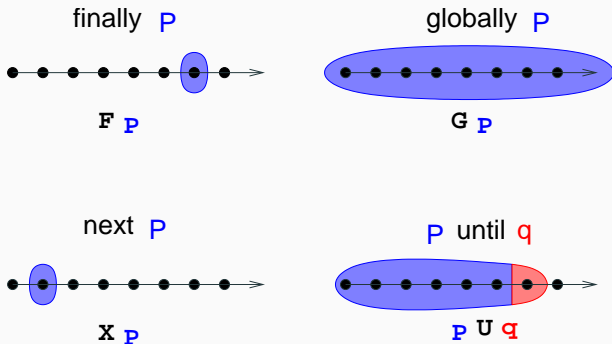
Example: modulo 4 counter with reset [2/2]

- The invariant is **false**

```
nuXmv > read_model -i counter4reset.smv;
nuXmv > go; check_invar
-- invariant out < 2 is false
...
-> State: 1.1 <-
  b0 = FALSE
  b1 = FALSE
  reset = FALSE
  out = 0
-> State: 1.2 <-
  b0 = TRUE
  out = 1
-> State: 1.3 <-
  b0 = FALSE
  b1 = TRUE
  out = 2
```

LTL specifications

- LTL properties are specified via the keyword `LTLSPEC:`
`LTLSPEC <ltl_expression>`



- LTL properties are checked via the `check_ltlspec` command

Specifications Examples:

- A state in which $\text{out} = 3$ is eventually reached

Specifications Examples:

- A state in which `out = 3` is eventually reached

`LTLSPEC F out = 3`

- Condition `out = 0` holds until `reset` becomes false

Specifications Examples:

- A state in which `out = 3` is eventually reached

LTLSPEC `F out = 3`

- Condition `out = 0` holds until `reset` becomes false

LTLSPEC `(out = 0) U (!reset)`

- Every time a state with `out = 2` is reached, a state with `out = 3` is reached afterward

Specifications Examples:

- A state in which $out = 3$ is eventually reached

LTLSPEC $F out = 3$

- Condition $out = 0$ holds until $reset$ becomes false

LTLSPEC $(out = 0) U (!reset)$

- Every time a state with $out = 2$ is reached, a state with $out = 3$ is reached afterward

LTLSPEC $G (out = 2 \rightarrow F out = 3)$

LTL specifications

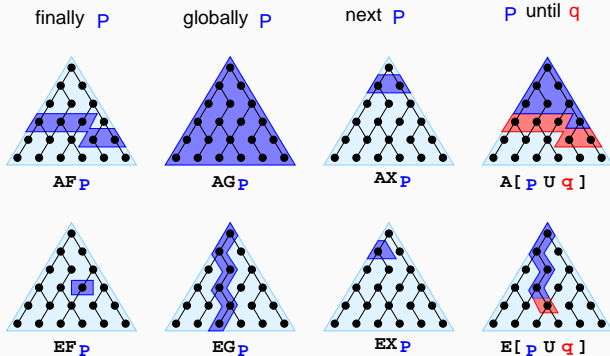
All the previous specifications are false:

```
NuSMV > check_ltlspec
-- specification F out = 3 is false ...
-- loop starts here --
-> State 1.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = TRUE
    out = 0
-> State 1.2 <-
-- specification (out = 0 U (!reset)) is false ...
-- loop starts here --
-> State 2.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = TRUE
    out = 0
-> State 2.2 <-
-- specification G (out = 2 -> F out = 3) is false ...
```

Q: why?

CTL specifications

- CTL properties are specified via the keyword CTLSPEC:
CTLSPEC <ctl_expression>



- CTL properties are checked via the `check_ctlspec` command

Specifications Examples:

- It is possible to reach a state in which $\text{out} = 3$

Specifications Examples:

- It is possible to reach a state in which $out = 3$
CTLSPEC EF $out = 3$
- It is inevitable that $out = 3$ is eventually reached

Specifications Examples:

- It is possible to reach a state in which $out = 3$
CTLSPEC EF $out = 3$
- It is inevitable that $out = 3$ is eventually reached
CTLSPEC AF $out = 3$
- It is always possible to reach a state in which $out = 3$

Specifications Examples:

- It is possible to reach a state in which $out = 3$
CTLSPEC EF $out = 3$
- It is inevitable that $out = 3$ is eventually reached
CTLSPEC AF $out = 3$
- It is always possible to reach a state in which $out = 3$
CTLSPEC AG EF $out = 3$
- Every time a state with $out = 2$ is reached, a state with $out = 3$ is reached afterward

Specifications Examples:

- It is possible to reach a state in which $out = 3$
CTLSPEC EF $out = 3$
- It is inevitable that $out = 3$ is eventually reached
CTLSPEC AF $out = 3$
- It is always possible to reach a state in which $out = 3$
CTLSPEC AG EF $out = 3$
- Every time a state with $out = 2$ is reached, a state with $out = 3$ is reached afterward
CTLSPEC AG ($out = 2 \rightarrow$ AF $out = 3$)
- The reset operation is correct

Specifications Examples:

- It is possible to reach a state in which $out = 3$
CTLSPEC EF $out = 3$
- It is inevitable that $out = 3$ is eventually reached
CTLSPEC AF $out = 3$
- It is always possible to reach a state in which $out = 3$
CTLSPEC AG EF $out = 3$
- Every time a state with $out = 2$ is reached, a state with $out = 3$ is reached afterward
CTLSPEC AG ($out = 2 \rightarrow$ AF $out = 3$)
- The reset operation is correct
CTLSPEC AG ($reset \rightarrow$ AX $out = 0$)

Fairness Constraints

The need for Fairness Constraints

The specification $\text{AF } \text{out} = 1$ is not verified

- On the path where **reset** is always **1**, the system loops on a state where **out = 0**:

```
reset = TRUE, TRUE, TRUE, TRUE, TRUE, ...  
out   = 0, 0, 0, 0, 0, 0, ...
```

Similar considerations for other properties:

- $\text{AF } \text{out} = 2$
- $\text{AF } \text{out} = 3$
- $\text{AG } (\text{out} = 2 \rightarrow \text{AF } \text{out} = 3)$
- ...

⇒ it would be **fair** to consider only paths in which the **counter** is not **reset** with such a high frequency so as to hinder its desired functionality

Fairness Constraints

NUXMV supports both *justice* and *compassion* fairness constraints

- **Fairness/Justice** p : consider only the executions that satisfy **infinitely often** the condition p
- **Strong Fairness/Compassion** (p, q) : consider only those executions that either satisfy p **finitely often** or satisfy q **infinitely often**
(i.e. p true infinitely often $\Rightarrow q$ true infinitely often)

Remarks:

- **verification**: properties must hold only on **fair paths**
- Currently, compassion constraints have some limitations (are supported only for BDD-based LTL model checking)

Example: modulo 4 counter with reset

Add the following fairness constraint to the model:

```
JUSTICE out = 3
```

(we consider only paths in which the counter reaches value 3 infinitely often)

All the properties are now verified:

```
nuXmv > reset
nuXmv > read_model -i counter4reset.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification EF out = 3 is true
-- specification AF out = 1 is true
-- specification AG (EF out = 3) is true
-- specification AG (out = 2 -> AF out = 3) is true
-- specification AG (reset -> AX out = 0) is true
```

Examples

Example: 4-bit adder [1/5]

We want to add a **request** operation to our adder, with the following semantics: every time a **request** is issued, the adder starts computing the sum of its operands. When finished, it stores the result in **sum**, setting **done** to true.

```
MODULE bit-adder(req, in1, in2, cin)
VAR
  sum: boolean;  cout: boolean;  ack: boolean;
ASSIGN
  init(ack) := FALSE;
  next(sum) := (in1 xor in2) xor cin;
  next(cout) := (in1 & in2) | ((in1 | in2) & cin);
  next(ack) := case
    req: TRUE;
    !req: FALSE;
  esac;
```

Example: 4-bit adder [2/5]

```
MODULE adder(req, in1, in2)
VAR
  bit[0]: bit-adder(
    req, in1[0], in2[0], FALSE);
  bit[1]: bit-adder(
    bit[0].ack, in1[1], in2[1],
    bit[0].cout);
  bit[2]: bit-adder(...);
  bit[3]: bit-adder(...);
DEFINE
  sum[0] := bit[0].sum;
  sum[1] := bit[1].sum;
  sum[2] := bit[2].sum;
  sum[3] := bit[3].sum;
  overflow := bit[3].cout;
  ack := bit[3].ack;
```

```
MODULE main
VAR
  req: boolean;
  a: adder(req, in1, in2);
ASSIGN
  init(req) := FALSE;
  next(req) :=
    case
      !req : {FALSE, TRUE};
      req :
        case
          a.ack : FALSE;
          TRUE: req;
        esac;
    esac;
DEFINE
  done := a.ack;
```


Example: 4-bit adder [3/5]

- Every time a `request` is issued, the adder will compute the `sum` of its operands

Example: 4-bit adder [3/5]

- Every time a `request` is issued, the adder will compute the sum of its operands

```
CTLSPEC AG (req -> AF sum = op1 + op2);
```

Example: 4-bit adder [3/5]

- Every time a `request` is issued, the adder will compute the sum of its operands

```
CTLSPEC AG (req -> AF sum = op1 + op2);
```

```
CTLSPEC AG (req -> AF (done &  
                        sum = op1 + op2));
```

Example: 4-bit adder [3/5]

- Every time a `request` is issued, the adder will compute the `sum` of its operands

```
CTLSPEC AG (req -> AF sum = op1 + op2);
```

```
CTLSPEC AG (req -> AF (done &  
                        sum = op1 + op2));
```

- Every time a `request` is issued, the `request` holds until the adder will compute the `sum` of its operands and set `done` to true

Example: 4-bit adder [3/5]

- Every time a `request` is issued, the adder will compute the sum of its operands

```
CTLSPEC AG (req -> AF sum = op1 + op2);
```

```
CTLSPEC AG (req -> AF (done &  
                    sum = op1 + op2));
```

- Every time a `request` is issued, the `request` holds until the adder will compute the sum of its operands and set `done` to true

```
CTLSPEC AG (req -> A[req U (done &  
                          (sum = op1 + op2))]);
```

Example: 4-bit adder [4/5]

```
nuXmv > read_model -i examples/4-adder-request.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification AG (req -> AF sum = op1 + op2) is false
-- as demonstrated by the following execution sequence
...
```

Issue: the adder circuit is unstable after first addition, `req` flips value due to a `.ack` still being true.

Example: 4-bit adder [5/5]

Fix

ASSIGN

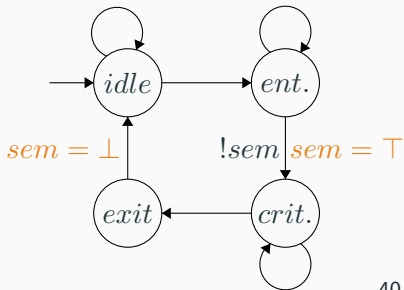
```
next(req) :=
  case
    !req:
      case
        !a.ack: {FALSE, TRUE};
        TRUE: req;
      esac;
    req:
      case
        a.ack : FALSE;
        TRUE: req;
      esac;
  esac;
```

Example: Simple Mutex [1/4]

```
MODULE user(semaphore)
VAR
  state : { idle, entering, critical, exiting };

ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : { idle, entering };
      state = entering & !semaphore : critical;
      state = critical : { critical, exiting };
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;
```

FAIRNESS
running



Example: Simple Mutex [2/4]

```
MODULE main
VAR
  semaphore : boolean;
  proc1      : process user(semaphore);
  proc2      : process user(semaphore);

ASSIGN
  init(semaphore) := FALSE;
```

Example: Simple Mutex [3/4]

Safety

two processes are never in the critical section at the same time

Example: Simple Mutex [3/4]

Safety

two processes are never in the critical section at the same time

```
CTLSPEC AG !(proc1.state = critical &
             proc2.state = critical);
```

Example: Simple Mutex [3/4]

Safety

two processes are never in the critical section at the same time

```
CTLSPEC AG !(proc1.state = critical &  
            proc2.state = critical);
```

Liveness

whenever a process is entering the critical section then sooner or later it will be in the critical section

Example: Simple Mutex [3/4]

Safety

two processes are never in the critical section at the same time

```
CTLSPEC AG !(proc1.state = critical &
             proc2.state = critical);
```

Liveness

whenever a process is entering the critical section then sooner or later it will be in the critical section

```
CTLSPEC AG (proc1.state = entering ->
            AF proc1.state = critical);
```

Example: Simple Mutex [4/4]

```
nuXmv > read_model -i examples/mutex_user.smv
nuXmv > go
nuXmv > check_ctlspec -n 0
-- specification AG !(proc1.state = critical &
                    proc2.state = critical) is true
```

Example: Simple Mutex [4/4]

```
nuXmv > read_model -i examples/mutex_user.smv
nuXmv > go
nuXmv > check_ctlspec -n 0
-- specification AG !(procl.state = critical &
                    proc2.state = critical)  is true

nuXmv > check_ctlspec -n 1
-- specification AG (procl.state = entering ->
                    AF procl.state = critical)  is false
...
```

Example: Simple Mutex [4/4]

```
nuXmv > read_model -i examples/mutex_user.smv
nuXmv > go
nuXmv > check_ctlspec -n 0
-- specification AG !(proc1.state = critical &
                    proc2.state = critical) is true

nuXmv > check_ctlspec -n 1
-- specification AG (proc1.state = entering ->
                    AF proc1.state = critical) is false
...
```

Issue

proc1 selected for execution only when proc2 is in critical section!

Example: Simple Mutex [4/4]

```
nuXmv > read_model -i examples/mutex_user.smv
nuXmv > go
nuXmv > check_ctlspec -n 0
-- specification AG !(proc1.state = critical &
                    proc2.state = critical) is true

nuXmv > check_ctlspec -n 1
-- specification AG (proc1.state = entering ->
                    AF proc1.state = critical) is false
...
```

Issue

proc1 selected for execution only when proc2 is in critical section!

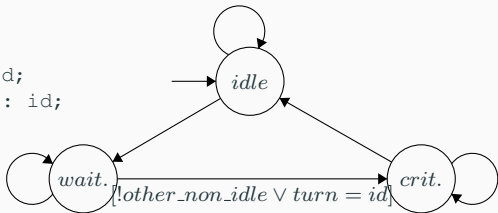
Fix

FAIRNESS

state = idle

Example: yet another mutex [1/4]

```
MODULE mutex(turn, other_non_idle, id)
VAR
  state: {idle, waiting, critical};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state=idle: {idle, waiting};
      state=waiting & (!other_non_idle|turn=id): critical;
      state=waiting: waiting;
      state=critical: {critical, idle};
    esac;
  next(turn) :=
    case
      next(state) = idle : !id;
      next(state) = critical : id;
      TRUE : turn;
    esac;
DEFINE
  non_idle := state in
    {waiting, critical};
FAIRNESS
  running
```



Example: yet another mutex [2/4]

```
MODULE main
VAR
  turn: boolean;
  p0: process mutex(turn,
                    p1.non_idle, FALSE);
  p1: process mutex(turn,
                    p0.non_idle, TRUE);
```

Example: yet another mutex [3/4]

Safety

Mutual exclusion

Example: yet another mutex [3/4]

Safety

Mutual exclusion

```
CTLSPEC AG !(p0.state=critical &  
             p1.state=critical)
```

Example: yet another mutex [3/4]

Safety

Mutual exclusion

```
CTLSPEC AG !(p0.state=critical &
             p1.state=critical)
```

Liveness

If someone wants to access the critical section then he/she will eventually succeed.

Example: yet another mutex [3/4]

Safety

Mutual exclusion

```
CTLSPEC AG !(p0.state=critical &
              p1.state=critical)
```

Liveness

If someone wants to access the critical section then he/she will eventually succeed.

```
CTLSPEC AG (p0.state=waiting ->
             AF (p0.state=critical))
```

Example: yet another mutex [4/4]

```
nuXmv > read_model -i mutex-another.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification AG !(p0.state = critical
                    & p1.state = critical) is true
```


Example: yet another mutex [4/4]

```
nuXmv > read_model -i mutex-another.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification AG !(p0.state = critical
                    & p1.state = critical) is true

-- specification AG (p0.state = waiting ->
                    AF p0.state = critical) is false
```

Example: yet another mutex [4/4]

```
nuXmv > read_model -i mutex-another.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification AG !(p0.state = critical
                    & p1.state = critical) is true

-- specification AG (p0.state = waiting ->
                    AF p0.state = critical) is false
```

Issue

A process can stay in critical section forever.

Example: yet another mutex [4/4]

```
nuXmv > read_model -i mutex-another.smv
nuXmv > go
nuXmv > check_ctlspec
-- specification AG !(p0.state = critical
                    & p1.state = critical) is true

-- specification AG (p0.state = waiting ->
                    AF p0.state = critical) is false
```

Issue

A process can stay in critical section forever.

Fix

```
FAIRNESS
    state=idle
```

Exercises

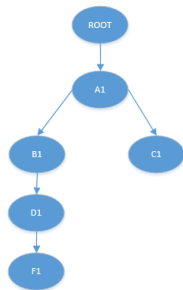
Exercises [1/2]

Simple Transition System

explain why all three properties are verified.

```
MODULE main
VAR
  state : {ROOT, A1, B1, C1, D1, F1, M1};

ASSIGN
  init(state) := ROOT;
  next(state) := case
    state = ROOT : A1;
    state = A1   : {B1, C1};
    state = B1   : D1;
    state = D1   : F1;
    TRUE        : state;
  esac;
```



```
CTLSPEC
```

```
AG( state=A1 -> AX ( A [ state=B1 U ( state=D1 -> EX state=F1 ) ] ) );
```

```
CTLSPEC
```

```
AG( state=A1 -> AX ( A [ state=B1 U ( state=F1 -> EX state=C1 ) ] ) );
```

```
CTLSPEC
```

```
AG( state=A1 -> AX ( A [ state=M1 U ( state=F1 -> EX state=C1 ) ] ) );
```

Bubblesort

implement a transition system which sorts the following input array $\{4, 1, 3, 2, 5\}$ with increasing order. Verify the following properties:

- there exists no path in which the algorithm ends
- there exists no path in which the algorithm ends with a sorted array

Bubblesort pseudocode

Bubblesort pseudocode

you might use the following *bubblesort pseudocode* as reference:

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```