nuXmv: property specification [*]

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/~trentin

Formal Methods Lab Class, Mar 31, 2015

Università degli Studi di Trento

[*] These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani for FM lab 2005/14

Comunications:

- no lesson on 7/04 (easter holiday)
- no lesson on 14/04 (teaching assistant afk)

- thus.. next lesson on 21/04

...take the chance to revise everything covered so far!

Contents

Property Specifications [1/2]

In NUXMV, property specifications:

- can be added to any module within a program
- can be specified through NUXMV interactive shell

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

- all properties are collected into an internal database, which can be visualized via the `show_property` command:

```
nuXmv > show_property
**** PROPERTY LIST [ Type, Status, Counter-example Number, Name ] ****
--------------------------  PROPERTY LIST  -------------------------
000 :AG !(proc1.state = critical & proc2.state = critical)
  [CTL          True          N/A    N/A]
001 :AG (proc1.state = entering -> AF proc1.state = critical)
  [CTL          True          N/A    N/A]
```

- Every property can be accessed through its database index:

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

Property Specifications [2/2]

Property verification:

- each property is separately verified
- the result is either “true” or “false”. In the latter case, a counterexample is generated
  - 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

Property Specifications [2/2]

Property verification:

- each property is separately verified
- the result is either "true" or "false". In the latter case, a counterexample is generated
  - 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:
  - properties on the reachable states (propositional formulas which must hold invariantly in the model)
    - *invariants* (`INVARSPEC`)
  - properties on the computation paths (*linear time* temporal logics):
    - LTL (`LTLSPEC`)
  - properties on the computation tree (*branching time* temporal logics):
    - CTL (`CTLSPEC`)

Invariant specifications

- Invariant properties are specified via the keyword `INVARSPEC`:

  ```
  INVARSPEC <simple_expression>
  ```

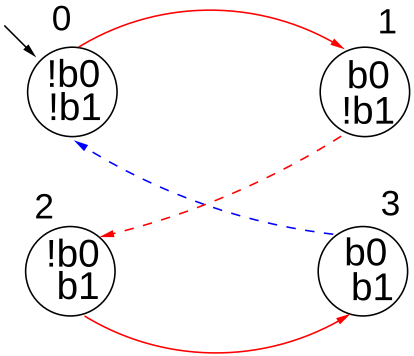
- Invariants are checked via the `check_invar` command

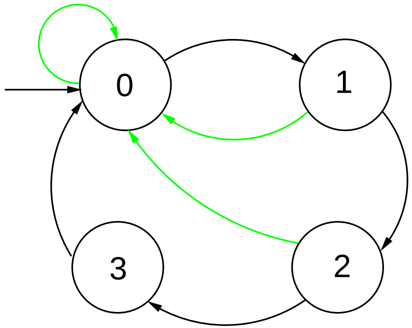
An example: the modulo 4 counter with reset [1/3]

```
MODULE main          -- counter4_reset.smv
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
```

An example: the modulo 4 counter with reset [2/3]





An example: the modulo 4 counter with reset [3/3]

The invariant is false

```
nuXmv > read_model -i counter4reset.smv
nuXmv > go
nuXmv > check_invar
-- invariant out < 2  is false
-- as demonstrated by the following execution sequence
Trace Description: AG alpha Counterexample
Trace Type: Counterexample
  -> 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>`

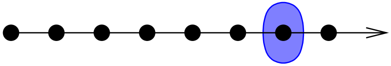
  where `<ltl_expression>` can contain the following temporal operators:

  `X _    F _    G _    _ U _`

- LTL properties are checked via the `check_ltlspec` command

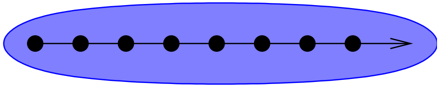
LTL specifications

finally P



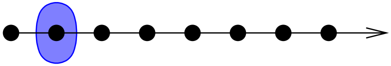
F P

globally P



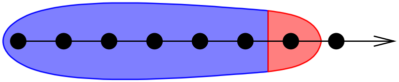
G P

next P



X P

P until q



P U q

LTL specifications

Specifications Examples:

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

LTL specifications

Specifications Examples:

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

```
LTLSPEC F out = 3
```

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

LTL specifications

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

LTL specifications

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 -> 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>
  ```

  where `<ctl_expression>` can contain the following temporal operators:

  ```
  AX _   AF _   AG _   A[_ U _]
  EX _   EF _   EG _   E[_ U _]
  ```

- CTL properties are checked via the `check_ctlspec` command

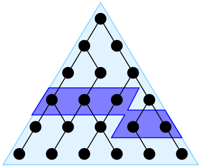
CTL specifications

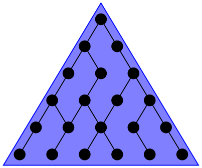
finally p

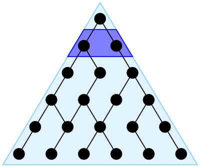
globally p

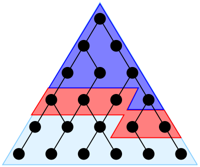
next p

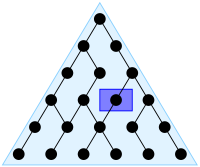
p until q

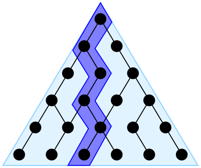

AF p

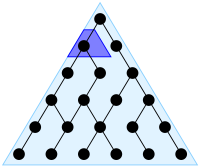

AG p

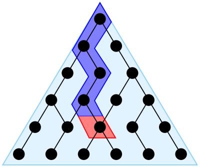

AX p


A[p U q]


EF p


EG p


EX p


E[p U q]

CTL specifications

Specifications Examples:

- It is possible to reach a state in which `out = 3`

CTL specifications

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

CTL specifications

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`

CTL specifications

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

CTL specifications

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 -> AF out = 3)
  ```

- The reset operation is correct

CTL specifications

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 -> AF out = 3)
  ```

- The reset operation is correct

  ```
  CTLSPEC AG (reset -> AX out = 0)
  ```

The need for Fairness Constraints

Let us consider again the counter with reset

- The specification `AF out = 1` is not verified
- On the path where `reset` is always `1`, the system loops on a state where `out = 0`, since the counter is always reset:

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

- Similar considerations hold for the property `AF out = 2`. For instance, the sequence

```
reset = FALSE,TRUE,FALSE,TRUE,FALSE,...
```

  generates the loop

```
out = 0,1,0,1,0,1...
```

  which is a counterexample to the given formula

Fairness Constraints

- It is desirable that certain conditions hold infinitely often
    - `AGAF p` is a fairness property
- Fairness conditions are used to eliminate behaviours in which a certain condition `p` never holds (i.e. ¬ `EFEG` ¬`p`)
- 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 ⇒ q true infinitely often)*

Remark:

- Currently, compassion constraints have some limitations (are supported only for BDD-based LTL model checking).

Fairness Constraints

Let us consider again the counter with reset. Let us add the following fairness constraint:

```
JUSTICE out = 3
```

(we restrict to paths in which the counter has value 3 infinitely often)

The following 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
```

The 4-bit adder example

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;
```

The 4-bit adder example

```
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;
```

The 4-bit adder example

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

The 4-bit adder example

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

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

The 4-bit adder example

- 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));
  ```

The 4-bit adder example

- 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 untill the adder will compute the `sum` of its operands and set `done` to true

The 4-bit adder example

- 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 untill the adder will compute the `sum` of its operands and set `done` to true

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

The 4-bit adder example

```
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.

The 4-bit adder example

```
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.

- Fix:

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

```
      req:
        case
          a.ack : FALSE;
          TRUE: req;
        esac;
    esac;
```

The simple mutex example

```
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
```

The simple mutex example

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

The simple mutex example

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

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

The simple mutex example

- two processes are never in the critical section at the same time
  ```
  CTLSPEC AG !(proc1.state = critical & proc2.state = critical); -- safety
  ```
- whenever a process is entering the critical section then sooner or later it will be in the critical section

The simple mutex example

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

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

- 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); -- liveness
```

The simple mutex example

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

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

- 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); -- liveness
  ```

```
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
```

The simple mutex example

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

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

- 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); -- liveness
  ```

```
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
...
```

The simple mutex example

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

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

```
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!

The simple mutex example

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

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

- 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); -- liveness
  ```

```
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
  ```

Another mutex example

```
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: 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
```

Another mutex example

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

nuXmv > read_model -i mutex.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 true
```

Another mutex example

Example: allow a process to stay in critical section for an arbitrary amount of time. Change the line

```
state=critical: idle;
```

into

```
state=critical: {critical, idle};
```

Another mutex example

Example: allow a process to stay in critical section for an arbitrary amount of time. Change the line

```
state=critical: idle;
```

into

```
state=critical: {critical, idle};
```

Now the second property becomes false:

```
nuXmv > reset
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
...
```

Another mutex example

Example: allow a process to stay in critical section for an arbitrary amount of time. Change the line

```
state=critical: idle;
```

into

```
state=critical: {critical, idle};
```

Now the second property becomes false:

```
nuXmv > reset
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: process can stay in critical section forever.

Another mutex example

Example: allow a process to stay in critical section for an arbitrary amount of time. Change the line

```
state=critical: idle;
```

into

```
state=critical: {critical, idle};
```

Now the second property becomes false:

```
nuXmv > reset
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: process can stay in critical section forever.

- Fix:

```
FAIRNESS
   state=idle
```

Another mutex example

The third property is still not verified:

```
nuXmv > check_ctlspec -n 2
-- specification AG !(p0.state = waiting & p1.state = waiting)  is false
...
```

Another mutex example

The third property is still not verified:

```
nuXmv > check_ctlspec -n 2
-- specification AG !(p0.state = waiting & p1.state = waiting)  is false
...
```

Issue: both processes can be temporarily both waiting (e.g. p0 waits first, p1 wait for second, and it's p0 turn)

Another mutex example

The third property is still not verified:

```
nuXmv > check_ctlspec -n 2
-- specification AG !(p0.state = waiting & p1.state = waiting)  is false
...
```

Issue: both processes can be temporarily both waiting (e.g. p0 waits first, p1 wait for second, and it's p0 turn)

- Fix: change the line

  ```
  state=waiting & (!other_non_idle|turn=id): critical;
  ```

  into

  ```
  state=waiting & (!other_non_idle): critical;
  ```

  and get

  ```
  nuXmv > check_ctlspec -n 2
  -- specification AG !(p0.state = waiting & p1.state = waiting)  is true
  ```