Formal Method Mod. 2 (Model Checking)

Laboratory 8

Giuseppe Spallitta
giuseppe.spallitta@unitn.it

Università degli studi di Trento

May 8, 2024

Outline

Model Properties [1/2]

A property:

- can be added to any module within a program

```
LTLSPEC G (req -> F sum = op1 + op2);
```

- can be specified through nuXmv interactive shell

```
nuXmv > check_ltlspec -p "G (req -> F 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 : G !(proc1.state = critical & proc2.state = critical)
  [LTL            True            N/A     N/A]
001 : G (proc1.state = entering -> F proc1.state = critical)
  [LTL            True            N/A     N/A]
```

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

```
nuXmv > check_ltlspec -n 0
```

Model Properties [2/2]

Property verification:

- each property is separately verified
- the result is either "TRUE" or "FALSE + counterexample"

Different kinds of properties are supported:

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

Invariants

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

  `INVARSPEC <simple_expression>`
- Invariants are checked via the `check_invar` command

  Remark:

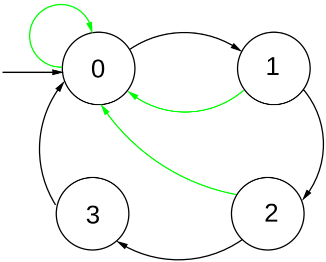
  during the checking of invariants, all the fairness conditions associated with the model are ignored

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]

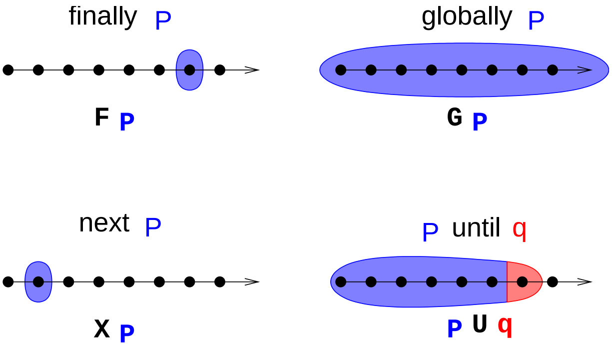
- The invariant is false

```
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

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

UNIVERSITÀ DEGLI STUDI DI TRENTO

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:

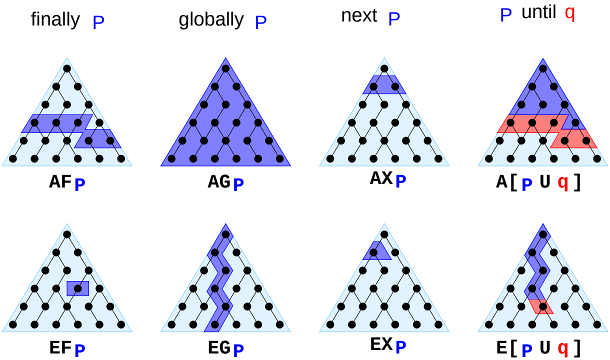
```
NuSMV > check_ltlspec
-- specification  F out = 3 is false ...
-- loop starts here --
-> State 1.1 <-
    b0 = FALSE
    b1 = FALSE
    reset = TRUE
    out = 0
-> State 1.2 <-
-- specification (out = 0 U (!reset)) is false ...
-- loop starts here --
-> State 2.1 <-
    b0 = FALSE
    b1 = FALSE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



- CTL properties are checked via the `check_ctlspec` command

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

UNIVERSITÀ DEGLI STUDI DI TRENTO

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

UNIVERSITÀ DEGLI STUDI DI TRENTO

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)`

Outline

The need for Fairness Constraints

The specification `F out = 3` 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:

- `F out = 1`
- `F out = 2`
- `G (out = 2 -> F out = 3)`
- ...

$\Longrightarrow$ 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_ltlspec
-- specification F out = 1  is true
-- specification G (out = 2 -> F out = 3)  is true
-- specification G (reset -> F out = 0)  is true
```

Outline

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
}
```

Main idea

- We will define a program counter pc that stores the current status of the execution (i.e. the line we reached).
- According to the iterative and conditional cycle, the program counter and the variables (when required) will change.

Example: model programs in nuXmv [2/4]

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

```
    void main() {
        ... // initialization of a and b
l1:     while (a!=b) {
l2:         if (a>b)
l3:             a=a-b;
            else
l4:             b=b-a;
        }
l5:     ... // 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: {l1,l2,l3,l4,l5};
ASSIGN
  init(pc):=l1;
  next(pc):=
    case
      pc=l1 & a!=b   : l2;
      pc=l1 & a=b    : l5;
      pc=l2 & a>b    : l3;
      pc=l2 & a<=b   : l4;
      pc=l3 | pc=l4  : l1;
      pc=l5          : l5;
    esac;
```

```
next(a):=
  case
    pc=l3 & a > b: a - b;
    TRUE: a;
  esac;

next(b):=
  case
    pc=l4 & b >= a: b-a;
    TRUE: b;
  esac;
```

Model programs in nuXmv: properties

- Let's check if, given $a = 16$ and $b = 12$, then we will eventually get as a result 4.

```
LTLSPEC (a = 16 & b = 12) -> F (a = 4 & b = 4)
```

- Let's check if both numbers will never reach negative values:

```
INVARSPEC a > 0 & b > 0
```

Example: model programs in nuXmv [4/4]

Step 2: **(alternative)**: use the constraint style

```
MODULE main
VAR
  a : 0..100;  b : 0..100;  pc : {l1, l2, l3, l4, l5};
INIT pc = l1
TRANS
  pc = l1 -> (((a != b & next(pc) = l2) |
               (a = b & next(pc) = l5)) &
              next(a) = a & next(b) = b)
TRANS
  pc = l2 -> (((a > b & next(pc) = l3) |
               (a < b & next(pc) = l4)) &
              next(a) = a & next(b) = b)
TRANS
  pc = l3 -> (next(pc) = l1 & next(a) = (a - b) & next(b) = b)
TRANS
  pc = l4 -> (next(pc) = l1 & next(b) = (b - a) & next(a) = a)
TRANS
  pc = l5 -> (next(pc) = l5 & next(a) = a & next(b) = b)
```

Outline

1. Model Properties

2. Fairness Constraints

3. Modelling a Program in nuXmv

4. Examples
   - The snail dungeon
   - Chimical reactions

5. Homework

The snail dungeon

You want to simulate the gameplay of "the snail dungeon":

- You have a path with 10 cells, with 2 good and 2 bad teleports. Encode a variable "turn" whose value could be {DICE, GOOD, BAD}, and a variable "steps" counting how many times a dice has been thrown. Once set, the teleport positions remain fixed.
- Each turn you throw a 3d-dice and move to the designated cell of the grid. If it is empty, you move on to the next turn. Notice that the dice is karmic, i.e. there is no way I get the same number from the dice from two consecutive throws.
- If you get into a good teleport, the next value of turn will be GOOD and you will move onward by 2 cells without increasing steps.
- If you get into a good teleport, the next value of turn will be BAD and you will move back by 2 cells without increasing steps.

Encode the gameplay of this game using nuXmv and check if there is a run of the game when I can win with less than 2 steps.

Science modeling

UNIVERSITÀ DEGLI STUDI DI TRENTO

Assume the following chemical reactions hold:

$$2O \rightarrow O_2$$

$$C + O \rightarrow CO$$

$$2C + O_2 \rightarrow 2CO$$

$$C + O_2 \rightarrow CO_2$$

Given a certain number of input carbon and oxygen atoms, is there any way for the contents of his reaction vessel to progress to a state where it contains three molecules of CO2? Model the contents of the reaction vessel in NuSMV.

Science modeling (cont.d)

- We can store the number of current atoms/molecules for each iteration using bounded integers.
- An enumerate variable can be used to define what reaction should be considered in the next step, ensuring non-determinism when necessary.

Science modeling (cont.d)

UNIVERSITÀ DEGLI STUDI DI TRENTO

```
MODULE main
    VAR
        o : 0..32;
        o2: 0..32;
        c : 0..32;
        co : 0..32;
        co2 : 0..32;
        reaction : {r1, r2, r3, r4, none};

    ASSIGN
        init(o) := 6;
        init(c) := 6;
        init(co) := 0;
        init(co2) := 0;
        init(o2) := 0;
        init(reaction) := none;
```

Science modeling (cont.d)

Transitions to define the next reaction that will take place on the next step.

```
TRANS
    (next(o) < 2) -> (next(reaction) != r1)
TRANS
    (next(o) < 1 | next(c) < 1) -> (next(reaction) != r2)
TRANS
    (next(o2) < 1 | next(c) < 2) -> (next(reaction) != r3)
TRANS
    (next(o2) < 1 | next(c) < 1) -> (next(reaction) != r4)
```

Science modeling (cont.d)

Transitions to define the new values for each molecule after a reaction took place.

```
TRANS
    (reaction = none) -> (o = next(o) & o2 = next(o2) &
        c = next(c) & co = next(co) & co2 = next(co2))

TRANS
    (reaction = r1) -> (next(o) = o - 2 & next(o2) = o2 + 1 &
        next(c) = c & next(co) = co & next(co2) = co2)

TRANS
    (reaction = r2) -> (next(o) = o - 1 & next(o2) = o2 &
        next(c) = c - 1 & next(co) = co + 1 & next(co2) = co2)

TRANS
    (reaction = r3) -> (next(o) = o & next(o2) = o2 - 1 &
        next(c) = c - 1 & next(co) = co + 2 & next(co2) = co2)

TRANS
    (reaction = r4) -> (next(o) = o & next(o2) = o2 - 1 &
         next(c) = c - 1 & next(co) = co & next(co2) = co2 + 1)
```

Science modeling: property

- If we are interested in knowing if there is a path that generates 3 CO_2 molecules, LTL apparently seems ineffective...
- ... but we can use it to search a valid counterproof that returns the desired execution.
- In this case we try to verify the number of $CO2$ molecules does not reach 3 in any path. If the property is not satisfied, the counterproof will returns a series of event reaching the condition.

Outline

1. Model Properties
2. Fairness Constraints
3. Modelling a Program in nuXmv
4. Examples
5. Homework

Homework

UNIVERSITÀ DEGLI STUDI DI TRENTO

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

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
```