Formal Method Mod. 2 (Model Checking)

Laboratory 8

Giuseppe Spallitta
giuseppe.spallitta@unitn.it

Università degli studi di Trento

May 5, 2021

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 (we won't see them in the laboratories).

Invariants

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

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

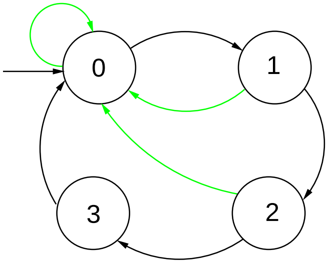
  Remark:
  during the checking of invariants, all the fairness conditions associated with the model are ignored

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]

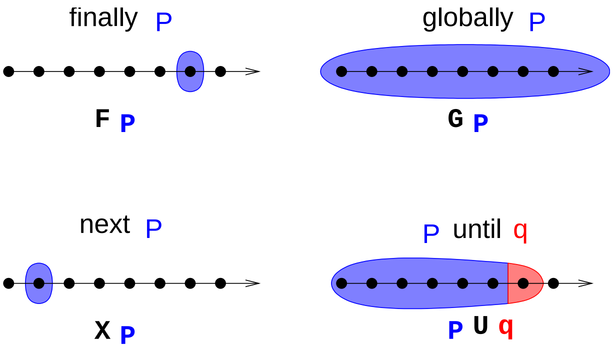
- The invariant is false

```
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

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?

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 ⇒ 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 number will never reach negative values:

```
INVARIANT 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

Mutual Exclusion

Two users U0 and U1, and an Arbiter Ar are part of a competition. Each user can be either `NonCritical`, `Trying` or `Critical`. To access the critical section, they notify their wish to the arbiter using 2 `req` variables, one per user. The arbiter notifies the possibility to access the resource using 2 `auth` variables. Moreover:

- From `NonCritical`, they can nondeterministically go to `Trying`;
- From `Trying`, they can go to `Critical` when authorized by the arbiter;
- From `Critical`, they can nondeterministically go back to `NonCritical`.

Model the problem on nuXvm and use LTL to encode the property "The aim of the arbiter is guaranteeing that the two users are not in status `Critical` at the same time"

A first attempt (cont.d)

```
MODULE User(auth)
  VAR
    status: { NonCritical, Trying, Critical };
    req: boolean;

  ASSIGN
    init(status) := NonCritical;
    next(status) :=
      case
        status = NonCritical : { NonCritical, Trying };
      status = Trying       :
            case
              next(auth) = FALSE : Trying;
              next(auth) = TRUE : Critical;
            esac;
        status = Critical     : { Critical, NonCritical};
      esac;

    req := status in { Trying, Critical };
```

A first attempt

```
MODULE Arbiter(req0, req1)
  VAR
    auth0: boolean;
    auth1: boolean;

  ASSIGN
    init(auth0) := FALSE;
    next(auth0) := req0 & !auth1;
    init(auth1) := FALSE;
    next(auth1) := req1 & !auth0;


MODULE main
  VAR
    U0: User(Ar.auth0);  --- User 0
    U1: User(Ar.auth1);  --- User 1

    Ar: Arbiter(U0.req, U1.req);

LTLSPEC G (!(U0.status = Critical & U0.status = Critical))
```

Fixing the issue

- You can see that the properties does not hold, and a counterproof is shown by the tool...
- We can define a variable `turn` defining the user that has the right to enter.
  - If user 0 is authorized to access the critical section, turn will be equal to 0.
  - If user 1 is authorized to access the critical section, turn will be equal to 1.
  - Otherwise, turn ranges cyclically on all the users to ensure fairness.

Fixing the issue (cont.d)

```
MODULE Arbiter(req0, req1)
  VAR
    auth0: boolean;
    auth1: boolean;
    turn: {0,1};

  ASSIGN
    init(auth0) := FALSE;
    next(auth0) := req0 & turn = 0;
    init(auth1) := FALSE;
    next(auth1) := req1 & turn = 1;
    next(turn)  := case
        next(auth0) : 0;
        next(auth1) : 1;
        TRUE : (turn+1) mod 2;
    esac;
```

Is fairness ensured?

- If we try to write a property to verify the two users have a fair access to the resource, you'll see it is not satisfied...

```
LTLSPEC G (U0.status = Trying ->
           F (U0.status = Critical))
```

- ... but we can easily solve the issue adding a `FAIRNESS` constraint to the model.

Science modeling

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)

```
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

Exercises [2/2]

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
```