UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Automated Reasoning and **Formal Verification**

Laboratory 8

Gabriele Masina
gabriele.masina@unitn.it
https://github.com/masinag/arfv2025

Università di Trento

April 30, 2025

These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin, Giuseppe Spallitta for FM lab 2005-2024.

Outline

Model Properties [1/2]

Specifying Properties

- directly **in the module**

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

- or **via nuXmv interactive shell**

```
nuXmv > check_ltlspec -p "G (req -> F sum = op1 + op2)"
```

- `show_property` lists all properties specified (in the module or via `add_property`):
- properties can be verified one at a time using its **database index**

```
nuXmv > show_property
**** PROPERTY LIST [ Type, Status, Counter-example Number, Name ] ****
--------------------------  PROPERTY LIST  --------------------------
000 :out < 2
  [Invar           False          1       N/A]
001 : G ( F out = 1)
  [LTL             Unchecked      N/A     N/A]
nuXmv > check_ltlspec -n 1
```

Model Properties [2/2]

Property verification:

- Each property is verified independently
- The result is either "`TRUE`" or "`FALSE` + counterexample"

Property types

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`:

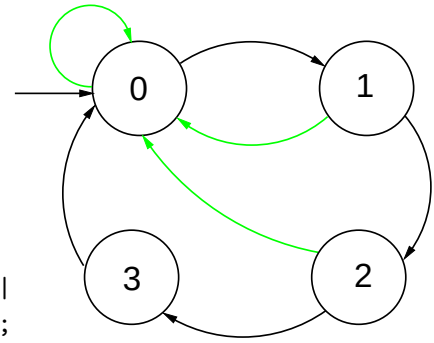
  `INVARSPEC <simple_expression>;`
- Invariants are checked via the `check_invar` command

Remark

When checking invariants, all the **fairness conditions** of 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;
```



Example: Modulo 4 Counter with Reset [2/2]

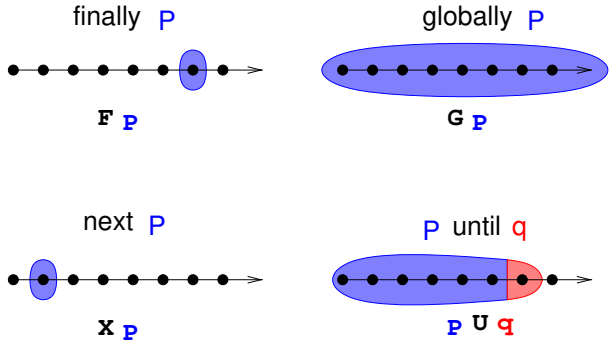
- The invariant is false

```
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);  -- False: reset can be true forever
LTLSPEC (!reset)  V (out = 0); -- True (V stands for "release")
```

- 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);  -- False: reset can be true forever
LTLSPEC (!reset)  V (out = 0); -- True (V stands for "release")
```

- 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

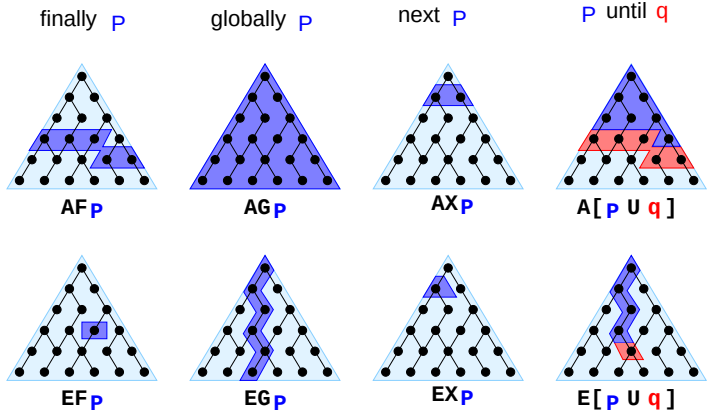
All the previous specifications are false:

```
nuXmv > 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

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);
```

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** $= \mathbf{0}$:
  `reset = TRUE, TRUE, TRUE, TRUE, TRUE, ...`
  `out = 0, 0, 0, 0, 0, ...`

Similar considerations for other properties:

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

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, ...
```

Similar considerations for other properties:

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

Fairness

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 `p` **infinitely often**

Strong Fairness: `COMPASSION (<p>, <q>)` consider only those executions that either satisfy `p` **finitely often** or satisfy `q` **infinitely often**
(i.e., p is true infinitely often $\Longrightarrow$ q is true infinitely often)

Fairness Constraints

nuXmv supports both **justice** and **compassion** fairness constraints:

Fairness: `JUSTICE <p>` consider only the executions that satisfy p **infinitely often**

Strong Fairness: `COMPASSION (<p>, <q>)` consider only those executions that either satisfy p **finitely often** or satisfy q **infinitely often**
(i.e., p is true infinitely often $\Longrightarrow$ q is true infinitely often)

Remarks

- For **verification**, properties must hold only on **fair paths**
- When checking invariants, all the fairness conditions are **ignored**.
- Currently, compassion constraints have some limitations, since they 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 **code**, how can we **model** and **verify** it with nuXmv?

```
def gcd(a: int, b: int) -> int:
  while a != b:
    if a > b:
      a = a-b
    else:
      b = b-a
  return a;  # at this point: GCD=a=b
```

Main idea

- We define a program counter pc that stores the current status of the execution (i.e., the line we reached).
- According to cycle and the conditional instructions, 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 with the line number

```
def gcd(a: int, b: int) -> int:
  while a != b:   # l1
    if a > b:     # l2
      a = a-b     # l3
    else:
      b = b-a     # l4
  return a;       # l5 -- at this point: 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;
```

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);
```

Model programs in nuXmv: properties

Step 3: check the properties of the program

- 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;
  ```

Outline

Exercise 8.1

Assume the following chemical reactions hold:

$$
\begin{array}{rcrcr}
 & & 2O & \rightarrow & O_2 \\
C & + & O & \rightarrow & CO \\
2C & + & O_2 & \rightarrow & 2CO \\
C & + & O_2 & \rightarrow & CO_2
\end{array}
$$

Given 6 carbon atoms and 6 oxygen atoms, is there any way for the contents of this reaction vessel to progress to a state where it contains three molecules of CO2? Model the contents of the reaction vessel in nuXmv.

Science Modelling: variables [1/2]

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

Science Modelling: variables [2/2]

```
MODULE main
VAR
  o         : 0..32;
  o2        : 0..32;
  c         : 0..32;
  co        : 0..32;
  co2       : 0..32;
  r : {r1, r2, r3, r4, none};
ASSIGN
  init(o)         := 6;
  init(c)         := 6;
  init(co)        := 0;
  init(co2)       := 0;
  init(o2)        := 0;
  init(r) := none;
```

Science Modelling: transitions [1/2]

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 Modelling: transitions [2/2]

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

```
TRANS
  r=none -> (next(o)  = o      & next(o2)  =  o2     & next(c) = c     &
             next(co) = co     & next(co2) = co2)
TRANS
  r=r1   -> (next(o)  =  o - 2 & next(o2)  =  o2 + 1 & next(c) = c     &
             next(co) = co     & next(co2) = co2);
TRANS
  r=r2   -> (next(o)  =  o - 1 & next(o2)  =  o2     & next(c) = c - 1 &
             next(co) = co + 1 & next(co2) = co2);
TRANS
  r=r3   -> (next(o)  = o      & next(o2)  =  o2 - 1 & next(c) = c - 1 &
             next(co) = co + 2 & next(co2) = co2);
TRANS
  r=r4   -> (next(o)  = o      & next(o2)  =  o2 - 1 & next(c) = c - 1 &
             next(co) = co     & next(co2) = co2 + 1);
```

Science Modelling: 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 counter-example corresponding to the desired execution.
- We try to verify that the number of $CO2$ molecules does not reach 3 in any path:
  `LTLSPEC G (co2!=3)`
- If the property is not satisfied, we get a sequence of steps reaching a state where co2=3.

The Snail Dungeon

Exercise 8.2

You want to simulate the gameplay of "**The Snail Dungeon**":

- You have a path with 10 cells, with 2 good and 2 bad teleports. Use 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 3-valued dice** and **move forward** the designated number of cells. If the arrival cell is empty, you move on to the next turn. Notice that the dice **cannot get the same number 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 *bad* teleport, the next value of turn will be `BAD` and you will **move back by 2 cells without increasing** `steps`.

Encode the game in nuXmv and find a way to reach position 10 with less than 3 steps.

The Snail Dungeon: variables

- The text already indicates some variables. Notice that with the default **BDD-based engine** we can only use **bounded integers**.

```
VAR
  pos   : 0..10;
  turn  : {DICE, GOOD, BAD};
  steps : 0..100;
  dice  : 1..3;
```

- We can use two arrays of two integers in `1..10` to store the positions of teleports.

```
good : array 0..1 of 1..10; bad  : array 0..1 of 1..10;
```

- These arrays are **fixed**. Instead of adding

```
ASSIGN next(good[0]) := good[0]; next(good[1]) := good[1]; ...
```

we can declare them as `FROZENVAR`:

```
FROZENVAR good : array 0..1 of 1..10;
          bad  : array 0..1 of 1..10;
```

The Snail Dungeon: initial values

- Initialize the game:

```
ASSIGN
  init(pos)   := 1;
  init(steps) := 0;
  init(turn)  :=DICE;
```

- And constrain the teleport positions:

```
INIT
  good[0] != good[1] & bad[0]  != bad[1] &
  good[0] != bad[0]  & good[0] != bad[1] &
  good[1] != bad[0]  & good[1] != bad[1];
```

The Snail Dungeon: transitions [1/2]

We encode the moves' logic:

```
ASSIGN
  next(turn) := case
    next(pos) = good[0] | next(pos) = good[1] : GOOD;
    next(pos) = bad[0]  | next(pos) = bad[1]  : BAD;
    TRUE: DICE;
  esac;
  next(steps) := case
    turn = DICE : min(steps + 1, 100);
    TRUE        : steps;
  esac;
  next(pos) := case
    turn = DICE : min(pos + dice, 10);
    turn = GOOD : min(pos + 2, 10);
    turn = BAD  : max(pos - 2, 0);
  esac;
```

Notice: we must ensure that bounded integers do not exceed their limits!

The Snail Dungeon: transitions [2/2]

Finally, we encode the dice logic: it cannot give the same value twice in a row.

```
TRANS
  (turn  = DICE -> next(dice) != dice) &
  (turn != DICE -> next(dice)  = dice);
```

The Snail Dungeon: properties

To get a trace where we reach position 10 with less than 3 steps, can negate the property and get a counter-example:

```
INVARSPEC pos=10 -> steps >= 3;
```

```
Trace Type: Counterexample
-> State: 1.1 <-
  good[0] = 8
  good[1] = 4
  bad[0] = 10
  bad[1] = 2
  pos = 1
  turn = DICE
  steps = 0
  dice = 3
-> State: 1.2 <-
  pos = 4
  turn = GOOD
  steps = 1
  dice = 2
-> State: 1.3 <-
  pos = 6
  turn = DICE
-> State: 1.4 <-
  pos = 8
  turn = GOOD
  steps = 2
  dice = 1
-> State: 1.5 <-
  pos = 10
  turn = BAD
```

Outline

Homework 8.1: 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

You might use the following *bubblesort* code as reference:

```
def bubbleSort(A : list[int]) -> None:
  n = len(A)
  swapped = True
  while swapped:
    swapped = false
    for i in range(1, n):
      if A[i-1] > A[i]: # if this pair is out of order
        # swap and remember something changed
        A[i-1], A[i] = A[i], A[i-1]
        swapped = true
```