UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Automated Reasoning and **Formal Verification**

Laboratory 10

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

Università di Trento

May 14, 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

1. Bounded Model Checking
2. k-Induction
3. Exercises

Bounded Model Checking

Idea

- Look for a **counter-example** path of increasing length k
  - **bug oriented:** *is there a bad behaviour?*

Bounded Model Checking

Idea

- Look for a **counter-example** path of increasing length k
  - **bug oriented:** *is there a bad behaviour?*
- For each k: build a Boolean formula that is satisfiable iff there is a counter-example of length k
  (can be expressed using $k \cdot |s|$ variables)

Bounded Model Checking

Idea

- Look for a **counter-example** path of increasing length k
  - **bug oriented:** *is there a bad behaviour?*
- For each k: build a Boolean formula that is satisfiable iff there is a counter-example of length k
  (can be expressed using $k \cdot |s|$ variables)
- Use of a **SAT/SMT procedure** to check the satisfiability of the Boolean formula
  - Can manage complex formulas on several variables (up to $\approx 10^5$)
  - Returns a satisfying assignment (i.e. a counter-example)

Commands for Bounded Model Checking

NuSMV / nuXmv

`go_bmc` initializes the system for the BMC verification with MiniSAT as backend.

`bmc_pick_state`, `bmc_simulate [-k]` simulate the system

`check_ltlspec_bmc` checks LTL specifications

`check_invar_bmc` checks INVAR specifications

nuXmv only

`go_msat` initializes the system to use MathSAT as backend

`msat_pick_state`, `msat_simulate [-k]` simulate the system

`msat_check_ltlspec_bmc` checks LTL specifications

`msat_check_invar_bmc` checks INVAR specifications

Example: BMC simulation

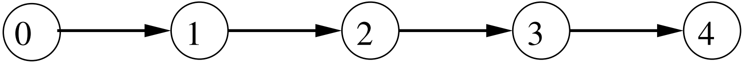
Modulo 8 Counter

```
MODULE main
VAR b0 : boolean; b1 : boolean;
    b2 : boolean;
INIT !b0 & !b1 & !b2;
ASSIGN
  next(b0) := !b0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
  next(b2) := ( (b0 & b1) & !b2) |
              (!(b0 & b1) &  b2);
DEFINE
  out  := 1*toint(b0) + 2*toint(b1) +
          4*toint(b2);
```

```
nuXmv > read_model -i counter8.smv
nuXmv > go_bmc; bmc_pick_state;
nuXmv > bmc_simulate -k 3 -p
-> State: 1.1 <-
b0 = FALSE
b1 = FALSE
b2 = FALSE
out = 0
-> State: 1.2 <-
b0 = TRUE
out = 1
-> State: 1.3 <-
b0 = FALSE
b1 = TRUE
out = 2
-> State: 1.4 <-
b0 = TRUE
out = 3
```

The following specification is **false**:

```
LTLSPEC G (out = 3 -> X out = 5);
```



- It is an example of **safety** property: *nothing bad ever happens*.
  - the counterexample is a **finite** trace (of length 4)
  - **important:** there are no counterexamples of length up to 3

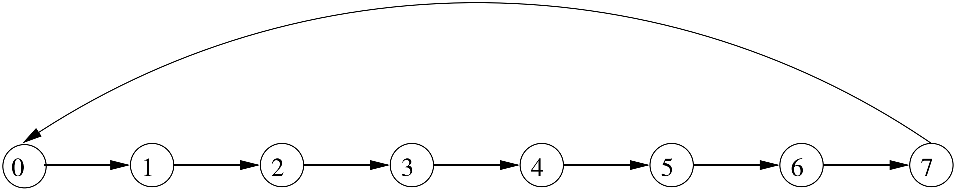
Checking LTL specifications: output

```
NuSMV > check_ltlspec_bmc -p "G (out = 3 -> X out = 5)"
-- no counterexample found with bound 0 for specification ...
-- no counterexample found with bound 1 for specification ...
-- no counterexample found with bound 2 for specification ...
-- no counterexample found with bound 3 for specification ...
-- specification  G (out = 3 ->  X out = 5)   is false
-- as demonstrated by the following execution sequence
-> State 1.1 <-
...
out = 0
-> State 1.2 <-
...
out = 1
-> State 1.3 <-
...
out = 2
-> State 1.4 <-
...
out = 3
-> State 1.5 <-
...
out = 4
```

Checking LTL specifications

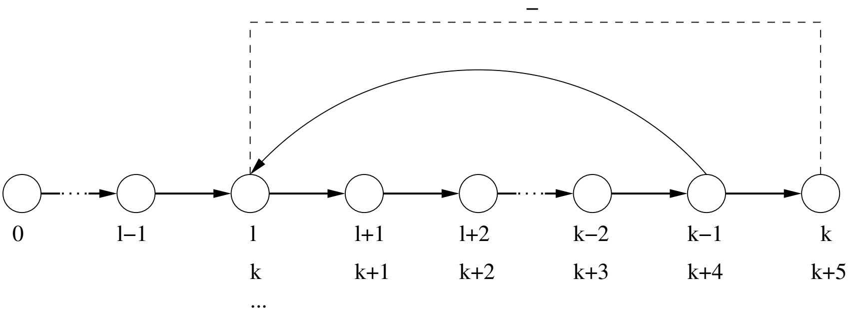
The following specification is false:

```
LTLSPEC ! G ( F   (out = 2));
LTLSPEC   F ( G ! (out = 2));
```



- It is an example of **liveness** property: *something desirable will eventually happen*
  - the counterexample is an **infinite** trace (*loop* of length 8)
  - since the state where `out = 2` is entered infinitely often, the property is **false**

Looping counterexample



- prefix : assignments from 0 to $l-1$,
- loop : infinitely repeat assignments l to $k-1$,
- loop-back : k^{th} assignment, always identical to l^{th} assignment.

Length and loopback condition

check_ltlspec_bmc looks for counterexamples of length up to k.

check_ltlspec_bmc_onepb looks for counterexamples of length exactly k.

- -l <bmc_loopback> to set the loopback conditions:
  - <bmc_loopback> >= 0 loop to a precise time point
  - <bmc_loopback> < 0 loop length
  - <bmc_loopback> = X no loopback
  - <bmc_loopback> = * all possible loopbacks (default)
- -k <bmc_length> to set the bounded length (default: 10)

set bmc_length <k> sets the default length to k

set bmc_loopback <l> sets the default loopback to l

Checking LTL specifications

Let us consider again the specification `! G ( F (out = 2))`

```
nuXmv > check_ltlspec_bmc_onepb -k 9 -l 0 -p "! G ( F (out = 2))"
-- no counterexample found with bound 9
   and loop at 0 for specification ...
```

Checking LTL specifications

Let us consider again the specification `! G ( F (out = 2))`

```
nuXmv > check_ltlspec_bmc_onepb -k 9 -l 0 -p "!  G ( F (out = 2))"
-- no counterexample found with bound 9
   and loop at 0 for specification ...

nuXmv > check_ltlspec_bmc_onepb -k 8 -l 1 -p "!  G ( F (out = 2))"
-- no counterexample found with bound 8
   and loop at 1 for specification ...
```

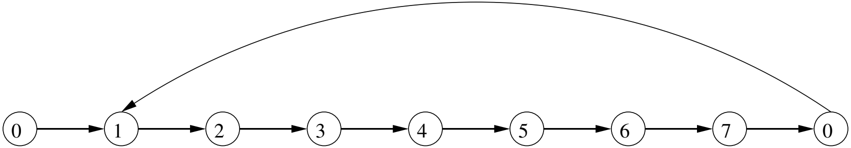
Checking LTL specifications

Let us consider again the specification `! G ( F (out = 2))`

```
nuXmv > check_ltlspec_bmc_onepb -k 9 -l 0 -p "! G ( F (out = 2))"
-- no counterexample found with bound 9
   and loop at 0 for specification ...

nuXmv > check_ltlspec_bmc_onepb -k 8 -l 1 -p "! G ( F (out = 2))"
-- no counterexample found with bound 8
   and loop at 1 for specification ...

nuXmv > check_ltlspec_bmc_onepb -k 9 -l 1 -p "! G ( F (out = 2))"
-- specification ! G  F out = 2   is false
-- as demonstrated by the following execution sequence
...
```



Checking LTL specifications

Let us consider again the specification `!G ( F (out =2))`

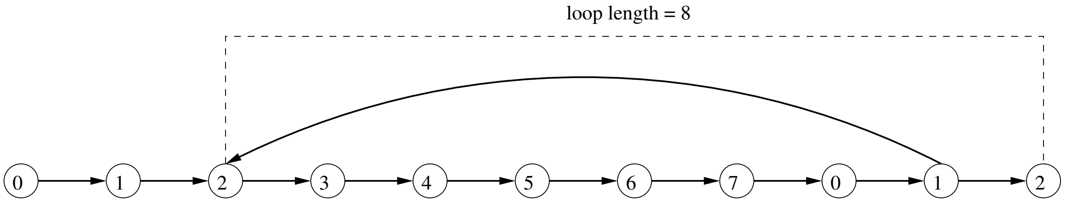
```
nuXmv > check_ltlspec_bmc_onepb -k 9 -l X -p "! G ( F (out =2))"
-- no counterexample found with bound 9 and no loop for specification ...
```

Checking LTL specifications

Let us consider again the specification !G (F (out =2))

```
nuXmv > check_ltlspec_bmc_onepb -k 9 -l X -p "! G ( F (out =2))"
-- no counterexample found with bound 9 and no loop for specification ...

nuXmv > check_ltlspec_bmc_onepb -k 10 -l -8 -p "! G ( F (out =2))"
-- specification ! G F out = 2  is false
-- as demonstrated by the following execution sequence
...
```



Outline

1. Bounded Model Checking
2. k-Induction
3. Exercises

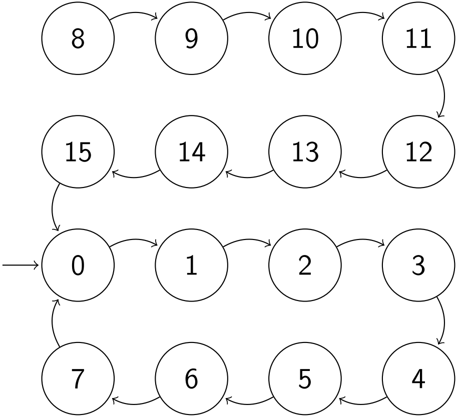
Checking invariants

- Bounded model checking can be used also for checking invariants
- Invariants are checked via the `check_invar_bmc` command
- Invariants are checked via an **inductive reasoning**, i.e. nuXmv tries to prove that:
  - the property **holds in** every **initial state**
  - the property **holds in** every state that is **reachable from** another state in which the property holds

Consider the following example:

```
MODULE main
VAR
  out : 0..15;
ASSIGN
  init(out) := 0;
  next(out) := case
    out = 7 :  0;
    TRUE    :  (out + 1) mod 16;
  esac;

INVARSPEC out in 0..10;
INVARSPEC out in 0..7;
```



Checking invariants

```
nuXmv > check_invar_bmc
-- cannot prove the invariant out in (0 .. 10) : the induction fails
-- as demonstrated by the following execution sequence
-> State 1.1 <-
out = 10
-> State 1.2 <-
out = 11
-- invariant out in (0 .. 7)   is true
```

- The invariant `out in 0..10` is **true**, but the induction **fails** because a state in which out=11 can be reached from a state in which out=10
  $\Longrightarrow$ if an invariant cannot be proved by inductive reasoning, it does not necessarily mean that the formula is false
- The stronger invariant `out in 0..7` is proved true by BMC, therefore also the invariant `out in 0..10` is true

Outline

1. Bounded Model Checking
2. k-Induction
3. Exercises
   - Cleaning Robot
   - Zip Puzzle
   - Homework

Exercise: Cleaning Robot [1/5]

Exercise 10.1: Cleaning Robot

Model a rechargeable cleaning robot which moves in a 10×10 room and cleans it. The robot state is composed of the following variables:

- `x`, `y` ranging in `0..9`, keep track of the robot's position;
- `state` with values in {`MOVE`, `CHECK`, `CHARGE`, `CLEAN`, `OFF`}, keeps track of the next action taken by the robot;
- `budget` ranging in `0..100`, signals the remaining power;
- `pos` output variable defined as `y*10 + x`.

Exercise: Cleaning Robot [1/5]

Exercise 10.1: Cleaning Robot

Model a rechargeable cleaning robot which moves in a 10×10 room and cleans it.
The robot state is composed of the following variables:

- `x`, `y` ranging in `0..9`, keep track of the robot's position;
- `state` with values in {`MOVE`, `CHECK`, `CHARGE`, `CLEAN`, `OFF`}, keeps track of the next action taken by the robot;
- `budget` ranging in `0..100`, signals the remaining power;
- `pos` output variable defined as `y*10 + x`.

```
MODULE main
VAR x : 0..9; y : 0..9; state : {MOVE, CHECK, CHARGE, CLEAN, OFF};
    budget : 0..100;
DEFINE pos := y*10 + x;
```

Exercise: Cleaning Robot [2/5]

Exercise 10.1: Cleaning Robot - initial state and budget

- At the beginning, the robot is in state `CHECK` and all other variables are 0.
- The budget is decreased by a single unit each time the robot is in state `MOVE` or `CLEAN` (and `budget > 0`)
- The budget is restored to 100 if the robot is in `CHARGE` state.
- Otherwise, the budget doesn't change.

Exercise: Cleaning Robot [2/5]

Exercise 10.1: Cleaning Robot - initial state and budget

- At the beginning, the robot is in state `CHECK` and all other variables are 0.
- The budget is decreased by a single unit each time the robot is in state `MOVE` or `CLEAN` (and `budget > 0`)
- The budget is restored to 100 if the robot is in `CHARGE` state.
- Otherwise, the budget doesn't change.

```
INIT state = CHECK & x = 0 & y = 0 & budget = 0;
ASSIGN
  next(budget) := case state in {MOVE,CLEAN} & budget > 0 : budget - 1;
                       state = CHARGE                     : 100;
                       TRUE                               : budget;
                  esac;
```

Exercise: Cleaning Robot [3/5]

Exercise 10.1: Cleaning Robot - `state` changes

The robot changes according to this **ordered** set of rules:

- if the robot is in `pos` 0 and `budget` is less than 100, then the next `state` is `CHARGE`
- if the `budget` is 0, then the next `state` is `OFF`
- if the robot is in `state` `CHARGE` or `MOVE`, then the next `state` is `CHECK`
- if the robot is in `state` `CHECK`, then the next `state` is either `CLEAN` or `MOVE`
- otherwise, the next `state` is `MOVE`.

Exercise 10.1: Cleaning Robot - state changes

The robot changes according to this **ordered** set of rules:

- if the robot is in pos 0 and budget is less than 100, then the next state is `CHARGE`
- if the budget is 0, then the next state is `OFF`
- if the robot is in state `CHARGE` or `MOVE`, then the next state is `CHECK`
- if the robot is in state `CHECK`, then the next state is either `CLEAN` or `MOVE`
- otherwise, the next state is `MOVE`.

```
next(state) :=  case pos = 0 & budget < 100  : CHARGE;
                  budget = 0                 : OFF;
                  state in {CHARGE, MOVE}    : CHECK;
                  state = CHECK              : {CLEAN, MOVE};
                  TRUE                       : MOVE;
                esac;
```

Exercise: Cleaning Robot [4/5]

Exercise 10.1: Cleaning Robot - moves

Encode, using the **constraint-style** (easier!), the following constraints:

- If the state is different than `MOVE`, then the position of the robot never changes.
- If the state is equal to `MOVE`, then the robot moves by a **single square** in one of the **cardinal directions**: either `x` or `y` changes, but not both at the same time.

Exercise: Cleaning Robot [4/5]

Exercise 10.1: Cleaning Robot - moves

Encode, using the **constraint-style** (easier!), the following constraints:

- If the state is different than `MOVE`, then the position of the robot never changes.
- If the state is equal to `MOVE`, then the robot moves by a **single square** in one of the **cardinal directions**: either `x` or `y` changes, but not both at the same time.

```
TRANS
  (state != MOVE -> (next(x) = x     & next(y) = y)) &
  (state  = MOVE -> (next(x) = x + 1 & next(y) = y    ) |
                    (next(x) = x - 1 & next(y) = y    ) |
                    (next(x) = x     & next(y) = y + 1) |
                    (next(x) = x     & next(y) = y - 1));
```

Exercise: Cleaning Robot [5/5]

Exercise 10.1: Cleaning Robot - properties

Encode and verify the following properties:

- In all possible executions, the robot changes position infinitely many times (false)
- It is never the case that the robot's action is either `MOVE` or `CLEAN` and the available budget is zero (false)
- If the robot charges infinitely often, then it changes position infinitely often (true)
- The robot does not move along the diagonals (true)

Exercise: Cleaning Robot [5/5]

Exercise 10.1: Cleaning Robot - properties

Encode and verify the following properties:

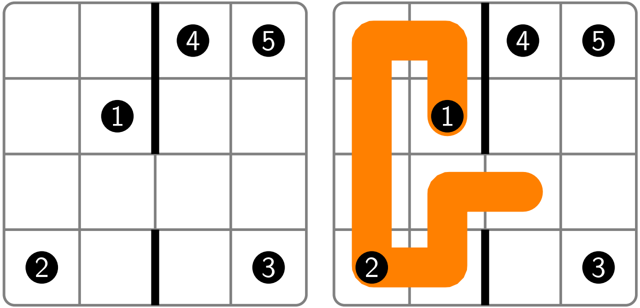
- In all possible executions, the robot changes position infinitely many times (false)
- It is never the case that the robot's action is either `MOVE` or `CLEAN` and the available budget is zero (false)
- If the robot charges infinitely often, then it changes position infinitely often (true)
- The robot does not move along the diagonals (true)

```
LTLSPEC   G F pos != next(pos);
LTLSPEC   G !(state in {MOVE, CLEAN} & budget = 0);
LTLSPEC   (G F state = CHARGE) -> (G F pos != next(pos));
INVARSPEC next(x) = x | next(y) = y;
```

Exercise: Zip Puzzle

Exercise 10.2: Zip Puzzle

Encode and solve the following puzzle as a planning problem using nuXmv.



1. Connect the first dot to the last.
2. Follow the order.
3. Move horizontally or vertically
4. Fill every cell
5. You cannot cross the walls (the bold lines)

Homework [1/3]

Homework 10.1: Cannibals

Three missionaries and three cannibals want to **cross a river** but they have only **one boat that holds two**.
If the cannibals ever **outnumber** the missionaries on either bank, the missionaries will be eaten.
The boat **cannot** cross the river by itself with **no people on board**.

The problem consists of finding a strategy to make them cross the river safely:

- Model the problem in SMV
- Use nuXmv to prove that there exists a solution to the planning problem

Homework [2/3]

Exercise 10.2: Gnome Sort [1/2]

Model the following code as a **module**:

```
def gnomeSort(arr: list[int], len: int) -> None:
    pos = 0                                               # 10
    while pos < len:                                      # 11
        if pos == 0 or arr[pos] >= arr[pos - 1]:          # 12
            pos = pos + 1                                 # 13
        else:
            arr[pos], arr[pos - 1] = arr[pos - 1], arr[pos]  # 14
            pos = pos - 1
    return  # self-loop here!                             # 15
```

Declare, inside the **main** module, the following variables:

- `arr`: array initialised to `{9, 7, 5, 3, 1}`
- `sorter`: instance of `gnomeSort(arr, 5)`

Homework [2/3]

Exercise 10.2: Gnome Sort [2/2]

Verify the following properties:

- The algorithm always terminates
- Eventually in the future, the array will be sorted forever
- Eventually the array is sorted, and the algorithm is not done until the array is sorted.

Homework [3/3]

Exercise 10.3: Leaping Frogs

The puzzle involves seven rocks and six frogs. The **seven rocks** are laid out in a **horizontal line** and the **six frogs** are divided into a **green trio** and a **brown trio**.

The green frogs sit on the rocks on the **right side** and the brown frogs sit on the rocks on the **left side**. The rock in the middle is vacant.

Can you swap the position of the two groups of frogs? Notice that you can only **move one frog at a time**, and they can **only move forward** to an empty rock or **jump over one (and only one) frog**, to reach an empty rock:

- Model the problem in SMV
- Use nuXmv find a solution to the planning problem