UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Automated Reasoning and **Formal Verification**

Laboratory 9

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

Università di Trento

May 07, 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

Planning Problem

Planning Problem

Given $\langle I, G, A \rangle$, where

I : (representation of) initial states

G : (representation of) goal states

A : (representation of) actions leading from one state to another

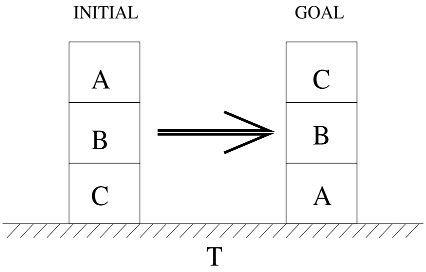
find a **plan** (sequence of actions) $a_1, \ldots, a_n$ leading from an initial state to a goal state.

Idea: from Planning to Model Checking

Encode a planning problem as a model checking problem:

1. Impose I as initial state(s)
2. Encode A as transition relation system
3. Verify the LTL property `(! F goal)`. The plan is a counter-example for it.

Example: Blocks [1/9]



$$
\begin{aligned}
&\mathit{Init}: && Above(A,B) \wedge Above(B,C) \wedge Above(C,T) \wedge Clear(A) \\
&\mathit{Goal}: && Above(C,B) \wedge Above(B,A) \wedge Above(A,T) \\
&Move(a,b,c) && \\
&\quad \mathit{Precond}: && Block(a) \wedge Clear(a) \wedge Above(a,b) \wedge (Clear(c) \vee Table(c)) \wedge \\
& && a \neq b \wedge a \neq c \wedge b \neq c \\
&\quad \mathit{Effect}: && Clear(b) \wedge \neg Above(a,b) \wedge Above(a,c) \wedge \neg Clear(c)
\end{aligned}
$$

Example: Blocks [2/9]

```
MODULE block(id, ab, bl)
VAR    above : {none, a, b, c}; -- the block above this one
       below : {none, a, b, c}; -- the block below this one
DEFINE clear := (above = none);
INIT   above = ab & below = bl
INVAR  below != id & above != id -- a block cannot be above/below itself

MODULE main
VAR    move : {move_a, move_b, move_c}; -- at each step one block moves
       block_a : block(a, none, b);
       block_b : block(b, a, c);
       block_c : block(c, b, none);
...
```

Example: Blocks [3/9]

- A block cannot move if it has some other block above itself

```
INVAR
  (!block_a.clear -> move != move_a) &
  (!block_b.clear -> move != move_b) &
  (!block_c.clear -> move != move_c);
```

Example: Blocks [3/9]

- A block cannot move if it has some other block above itself

```
INVAR
  (!block_a.clear -> move != move_a) &
  (!block_b.clear -> move != move_b) &
  (!block_c.clear -> move != move_c);
```

Q: Why `INVAR` and not `TRANS`?

Example: Blocks [3/9]

- A block cannot move if it has some other block above itself

```
INVAR
  (!block_a.clear -> move != move_a) &
  (!block_b.clear -> move != move_b) &
  (!block_c.clear -> move != move_c);
```

Q: Why `INVAR` and not `TRANS`?

A: `INVAR p` is equivalent to `INIT p` and `TRANS next(p)`.
With `TRANS` only, we could have in initial state with `move_b`.

Example: Blocks [3/9]

- A block cannot move if it has some other block above itself

```
INVAR
  (!block_a.clear -> move != move_a) &
  (!block_b.clear -> move != move_b) &
  (!block_c.clear -> move != move_c);
```

Q: Why `INVAR` and not `TRANS`?

A: `INVAR p` is equivalent to `INIT p` and `TRANS next(p)`.
With `TRANS` only, we could have in initial state with `move_b`.

Q: What's wrong with following formulation?

```
INVAR
  (block_a.clear -> move = move_a) &
  (block_b.clear -> move = move_b) &
  (block_c.clear -> move = move_c);
```

Example: Blocks [3/9]

- A block cannot move if it has some other block above itself

```
INVAR
  (!block_a.clear -> move != move_a) &
  (!block_b.clear -> move != move_b) &
  (!block_c.clear -> move != move_c);
```

Q: Why `INVAR` and not `TRANS`?

A: `INVAR p` is equivalent to `INIT p` and `TRANS next(p)`.
With `TRANS` only, we could have in initial state with `move_b`.

Q: What's wrong with following formulation?

```
INVAR
  (block_a.clear -> move = move_a) &
  (block_b.clear -> move = move_b) &
  (block_c.clear -> move = move_c);
```

A: Two `clear` blocks would cause an inconsistency (`move` can have **one** valid value).
Moreover, any `non-clear block` would still be able to `move`.

Example: Blocks [4/9]

- A moving `block` changes location and remains clear

```
TRANS
  (move = move_a -> next(block_a.clear) &
                    next(block_a.below) != block_a.below) &
  (move = move_b -> next(block_b.clear) &
                    next(block_b.below) != block_b.below) &
  (move = move_c -> next(block_c.clear) &
                    next(block_c.below) != block_c.below);
```

- A non-moving `block` does not change its location

```
TRANS
  (move != move_a -> next(block_a.below) = block_a.below) &
  (move != move_b -> next(block_b.below) = block_b.below) &
  (move != move_c -> next(block_c.below) = block_c.below);
```

Example: Blocks [5/9]

- A `block` remains connected to any *non-moving* `block`

```
TRANS
    (move != move_a & block_b.above = a -> next(block_b.above) = a) &
    (move != move_a & block_c.above = a -> next(block_c.above) = a) &
    (move != move_b & block_a.above = b -> next(block_a.above) = b) &
    (move != move_b & block_c.above = b -> next(block_c.above) = b) &
    (move != move_c & block_a.above = c -> next(block_a.above) = c) &
    (move != move_c & block_b.above = c -> next(block_b.above) = c);
```

Example: Blocks [5/9]

- A `block` remains connected to any *non-moving* `block`

```
TRANS
   (move != move_a & block_b.above = a -> next(block_b.above) = a) &
   (move != move_a & block_c.above = a -> next(block_c.above) = a) &
   (move != move_b & block_a.above = b -> next(block_a.above) = b) &
   (move != move_b & block_c.above = b -> next(block_c.above) = b) &
   (move != move_c & block_a.above = c -> next(block_a.above) = c) &
   (move != move_c & block_b.above = c -> next(block_b.above) = c);
```

Q: What about "`below`" block?

Example: Blocks [5/9]

- A `block` remains connected to any *non-moving* `block`

```
TRANS
   (move != move_a & block_b.above = a -> next(block_b.above) = a) &
   (move != move_a & block_c.above = a -> next(block_c.above) = a) &
   (move != move_b & block_a.above = b -> next(block_a.above) = b) &
   (move != move_b & block_c.above = b -> next(block_c.above) = b) &
   (move != move_c & block_a.above = c -> next(block_a.above) = c) &
   (move != move_c & block_b.above = c -> next(block_b.above) = c);
```

Q: What about "`below`" block?

A: Covered in previous slide!

Example: Blocks [6/9]

- Positioning of blocks: above and below relations must be symmetric.

```
INVAR
  (block_a.above = b <-> block_b.below = a) &
  (block_a.above = c <-> block_c.below = a) &
  (block_b.above = a <-> block_a.below = b) &
  (block_b.above = c <-> block_c.below = b) &
  (block_c.above = a <-> block_a.below = c) &
  (block_c.above = b <-> block_b.below = c);
```

Notice that the above handles also the case of `none` blocks!

Example: Blocks [7/9]

Remark

A **plan** is a sequence of transitions/actions leading from the initial state to an accepting/goal state.

Idea

- Assert property p: "goal state is not reachable"
- If a plan **exists**, nuXmv produces a counter-example for p
- The counterexample for p is a plan to reach the goal

Example: Blocks [8/9]

Examples

- Get a plan for reaching "goal state"

```
LTLSPEC
  ! F(block_a.below = none & block_a.above = b    &
      block_b.below = a    & block_b.above = c    &
      block_c.below = b    & block_c.above = none);
```

Example: Blocks [8/9]

Examples

- Get a plan for reaching "goal state"

```
LTLSPEC
  ! F(block_a.below = none & block_a.above = b    &
      block_b.below = a    & block_b.above = c    &
      block_c.below = b    & block_c.above = none);
```

- Get a plan for reaching a configuration in which all blocks are placed on the table

```
LTLSPEC
  ! F(block_a.below = none & block_b.below = none &
      block_c.below = none);
```

Example: Blocks [9/9]

Examples

- At any given time, at least one block is placed on the table

```
INVARSPEC
  block_a.below=none | block_b.below=none | block_c.below=none;
```

Examples

- At any given time, at least one block is placed on the table

```
INVARSPEC
  block_a.below=none | block_b.below=none | block_c.below=none;
```

- At any given time, at least one block has nothing above

```
INVARSPEC
  block_a.above=none | block_b.above=none | block_c.above=none;
```

Outline

1. Planning problem

2. Exercises
   - Tower of Hanoi
   - Ferryman
   - Tic-Tac-Toe

3. Homeworks

Tower of Hanoi [1/5]

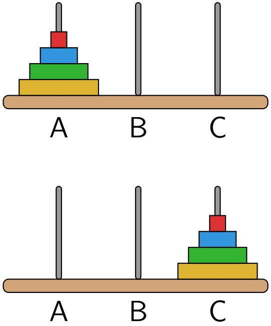
Exercise 9.1: Tower of Hanoi

Given 3 `poles` and N `disks` of different sizes:

Init A stack of disks with decreasing sizes is placed on pole A

Goal Move the stack to pole C

Rules
- Only one disk can be moved per step
- Only the upper disk can be moved
- A disk cannot be put above a smaller disk



Tower of Hanoi [2/5]

- Base system model

```
MODULE main
VAR d1   : {left,middle,right}; -- smallest
    d2   : {left,middle,right};
    d3   : {left,middle,right};
    d4   : {left,middle,right}; -- largest
    move : 1..4;                -- possible moves
```

Tower of Hanoi [2/5]

- Base system model

```
MODULE main
VAR d1   : {left,middle,right}; -- smallest
    d2   : {left,middle,right};
    d3   : {left,middle,right};
    d4   : {left,middle,right}; -- largest
    move : 1..4;                -- possible moves
```

- A disk cannot move if a smaller disk is above it (i.e. they share the same column)

```
DEFINE clear1 := TRUE;
       clear2 := d2 != d1;
       clear3 := d3 != d1 & d3 != d2;
       clear4 := d4 != d1 & d4 != d2 & d4 != d3;
```

Tower of Hanoi [3/5]

- Initial state

```
INIT d1 = left & d2 = left & d3 = left & d4 = left;
```

Tower of Hanoi [3/5]

- Initial state

```
INIT d1 = left & d2 = left & d3 = left & d4 = left;
```

- Moves descriptions

```
TRANS
  (move = 4 ->
    -- disks location changes
    next(d1) = d1 & next(d2) = d2 & next(d3) = d3 & next(d4) != d4 &
    -- d4 can not move on top of smaller disks
    next(d4) != d1 & next(d4) != d2 & next(d4) != d3) &
  (move = 3 -> ...)
  (move = 2 -> ...)
  (move = 1 -> ...);
```

Tower of Hanoi [4/5]

- A non-clear disk cannot move

```
INVAR
  (!clear1 -> move != 1) & (!clear2 -> move != 2) &
  (!clear3 -> move != 3) & (!clear4 -> move != 4);
```

Tower of Hanoi [4/5]

- A non-clear disk cannot move

```
INVAR
  (!clear1 -> move != 1) & (!clear2 -> move != 2) &
  (!clear3 -> move != 3) & (!clear4 -> move != 4);
```

- If all columns are being used, don't move the largest disk (or we would reach a deadlock).

```
INVAR
  ((clear1 & clear2 & clear3) -> move != 3) &
  ((clear1 & clear2 & clear4) -> move != 4) &
  ((clear1 & clear2 & clear3) -> move != 4) &
  ((clear2 & clear3 & clear4) -> move != 4);
```

Tower of Hanoi [5/5]

- Get a plan for a reaching "goal state":

```
LTLSPEC
  ! F (d1 = right & d2 = right & d3 = right & d4 = right);
INVARSPEC
  !   (d1 = right & d2 = right & d3 = right & d4 = right);
```

Ferryman [1/4]

Exercise 9.2: Ferryman

A ferryman has to bring a `sheep`, a `cabbage`, and a `wolf` safely across a river:

Init all the items are on the right side

Goal all the items are on the left side

Actions
- The ferryman can have at most one passenger on his boat
- The `cabbage` and the `sheep` cannot be left unattended on the same side of the river
- The `sheep` and the `wolf` cannot be left unattended on the same side of the river

Q: Can the ferryman transport all the items to the other side safely?

Ferryman [2/4]

- Base system model

```
MODULE main
VAR
  cabbage : {right,left};  sheep : {right,left};
  wolf    : {right,left};  man   : {right,left};
  move    : {c, s, w, e}; -- possible moves
DEFINE
  carry_cabbage := (move = c); carry_sheep := (move = s);
  carry_wolf    := (move = w); no_carry    := (move = e);
```

- Initial state

```
INIT cabbage = right & sheep = right & wolf = right & man = right;
```

Ferryman [3/4]

- The ferryman can move to the other side of the river with one item at a time

```
TRANS
  next(man) != man &
  (carry_cabbage ->
   (next(cabbage)!=cabbage & next(sheep) =sheep & next(wolf) =wolf)) &
  (carry_sheep ->
   (next(cabbage) =cabbage & next(sheep)!=sheep & next(wolf) =wolf)) &
  (carry_wolf ->
   (next(cabbage) =cabbage & next(sheep) =sheep & next(wolf)!=wolf)) &
  (no_carry ->
   (next(cabbage) =cabbage & next(sheep) =sheep & next(wolf) =wolf));
```

Ferryman [4/4]

- If the man is not in the same side of an item, we cannot choose it for the next movement (otherwise deadlock).

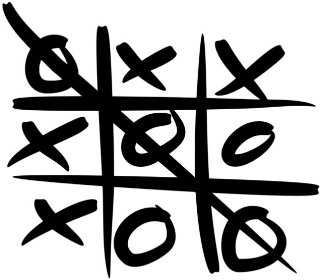
```
TRANS
  (next(man) != next(cabbage) -> !next(carry_cabbage)) &
  (next(man) != next(sheep)   -> !next(carry_sheep)) &
  (next(man) != next(wolf)    -> !next(carry_wolf));
```

- Get a plan for reaching a "goal state"

```
DEFINE
  safe_state := (sheep = wolf | sheep = cabbage) -> sheep = man;
  goal := cabbage = left & sheep = left & wolf = left;
LTLSPEC
  ! (safe_state U goal);
```

Exercise 9.3: Tic-Tac-Toe

Tic-tac-toe is a turn-based game for two adversarial players ($\times$ and $\circ$) marking the squares of a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical or diagonal row wins the game.



Scenario where $\circ$ wins

Tic-Tac-Toe [2/5]

- Base system model: we model the grid as an array of size 9:

```
  1 | 2 | 3
----|---|----
  4 | 5 | 6
----|---|----
  7 | 8 | 9
    |   |
```

```
MODULE main
VAR
  B : array 1..9 of {0,1,2};
  player : 1..2;
  move : 0..9;
```

Tic-Tac-Toe [2/5]

- Base system model: we model the grid as an array of size 9:

```
 1 | 2 | 3
___|___|___
 4 | 5 | 6
___|___|___
 7 | 8 | 9
   |   |
```

```
MODULE main
VAR
  B : array 1..9 of {0,1,2};
  player : 1..2;
  move : 0..9;
```

- Initial state

```
INIT
  B[1] = 0 & B[2] = 0 & B[3] = 0 & B[4] = 0 & B[5] = 0 &
  B[6] = 0 & B[7] = 0 & B[8] = 0 & B[9] = 0 & move = 0;
```

Tic-Tac-Toe [3/5]

- Turns Modeling

```
ASSIGN
  init(player) := 1;
  next(player) := player = 1 ? 2 : 1;
```

Tic-Tac-Toe [3/5]

- Turns Modeling

```
ASSIGN
  init(player) := 1;
  next(player) := player = 1 ? 2 : 1;
```

- Moves Modeling

```
TRANS
  B[1] != 0 -> next(move) != 1;
TRANS
  next(move) = 1 -> next(B[1]) = player &
    next(B[2])=B[2] & next(B[3])=B[3] & next(B[4])=B[4] &
    next(B[5])=B[5] & next(B[6])=B[6] & next(B[7])=B[7] &
    next(B[8])=B[8] & next(B[9])=B[9];
  ...
```

Tic-Tac-Toe [4/5]

- "End" state

```
DEFINE
  win1 := (B[1]=1 & B[2]=1 & B[3]=1) | (B[4]=1 & B[5]=1 & B[6]=1) |
          (B[7]=1 & B[8]=1 & B[9]=1) | (B[1]=1 & B[4]=1 & B[7]=1) |
          (B[2]=1 & B[5]=1 & B[8]=1) | (B[3]=1 & B[6]=1 & B[9]=1) |
          (B[1]=1 & B[5]=1 & B[9]=1) | (B[3]=1 & B[5]=1 & B[7]=1);
  win2 := (B[1]=2 & B[2]=2 & B[3]=2) | (B[4]=2 & B[5]=2 & B[6]=2) |
          (B[7]=2 & B[8]=2 & B[9]=2) | (B[1]=2 & B[4]=2 & B[7]=2) |
          (B[2]=2 & B[5]=2 & B[8]=2) | (B[3]=2 & B[6]=2 & B[9]=2) |
          (B[1]=2 & B[5]=2 & B[9]=2) | (B[3]=2 & B[5]=2 & B[7]=2);
  draw := !win1 & !win2 &
    !(B[1]=0 | B[2]=0 | B[3]=0 | B[4]=0 | B[5]=0 |
      B[6]=0 | B[7]=0 | B[8]=0 | B[9]=0);
INVAR
  (win1 | win2 | draw) <-> next(move)=0;
```

Tic-Tac-Toe [5/5]

- We can easily check if there is a way to reach every end state using the typical formulation:

```
LTLSPEC
    ! (F draw);
LTLSPEC
    ! (F win1);
LTLSPEC
    ! (F win2);
```

For each property, a trace satisfying the property is returned as a counterexample.

Outline

Homeoworks

Homework 9.1: Tower of Hanoi

Extend the Tower of Hanoi to handle five disks, and check that the goal state is reachable.

Homework 9.2: Ferryman

Another ferryman has to bring a fox, a chicken, a caterpillar and a crop of lettuce safely across a river.

Init All the items are on the right side

Goal All the items are on the left side

Rules

- The ferryman can cross the river with at most **two** passengers on his boat
- The fox eats the chicken if left unattended on the same side of the river
- The chicken eats the caterpillar if left unattended on the same side
- The caterpillar eats the lettuce if left unattended on the same side

Can the ferryman bring every item safely on the other side?