

NUXMV: Planning*

Patrick Trentin

`patrick.trentin@unitn.it`

`http://disi.unitn.it/trentin`

Formal Methods Lab Class, May 05, 2017



UNIVERSITÀ DEGLI STUDI DI
TRENTO

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi,
Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/16

Planning Problem

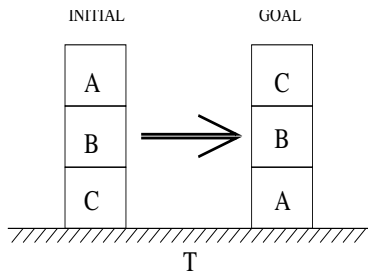
- **Planning Problem:** given $\langle I, G, T \rangle$, where
 - **I:** (representation of) initial state
 - **G:** (representation of) goal state
 - **T:** transition relation

find a sequence of transitions t_1, \dots, t_n leading from the initial state to the goal state.

- **Idea:** encode planning problem as a model checking problem
 - 1 impose **I** as initial state
 - 2 encode **T** as transition relation system
 - 3 verify the LTL property ! (**F goal_state**)

- 1 Planning problem
 - Blocks Example
- 2 Examples
 - The Tower of Hanoi
 - Ferryman
 - Tic-Tac-Toe
- 3 Exercises

Example: blocks [1/8]



Init : $On(A, B), On(B, C), On(C, T), Clear(A)$

Goal : $On(C, B), On(B, A), On(A, T)$

Move(a, b, c)

Precond : $Block(a) \wedge Clear(a) \wedge On(a, b) \wedge$

$(Clear(c) \vee Table(c)) \wedge$

$a \neq b \wedge a \neq c \wedge b \neq c$

Effect : $Clear(b) \wedge \neg On(a, b) \wedge$

$On(a, c) \wedge \neg Clear(c)$

Example: blocks [2/8]

```
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 can't be above or below itself

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

Example: blocks [3/8]

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

...

TRANS

```
(!block_a.clear -> move != move_a) &
```

```
(!block_b.clear -> move != move_b) &
```

```
(!block_c.clear -> move != move_c)
```

...

Example: blocks [3/8]

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

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

- **Q:** what's wrong with following formulation?

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

Example: blocks [3/8]

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

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

- **Q:** what's wrong with following formulation?

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

A:

- move can only have **one** valid value \implies **inconsistency** whenever there are two clear blocks at the same time
- any non-clear block would still be able to move

Example: blocks [4/8]

- 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 [5/8]

- 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 [5/8]

- 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 [5/8]

- 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!

- positioning of blocks is 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)

& (block_a.above = none -> (block_b.below != a & block_c.below != a))
& (block_b.above = none -> (block_a.below != b & block_c.below != b))
& (block_c.above = none -> (block_a.below != c & block_b.below != c))

& (block_a.below = none -> (block_b.above != a & block_c.above != a))
& (block_b.below = none -> (block_a.above != b & block_c.above != b))
& (block_c.below = none -> (block_a.above != c & block_b.above != c))
```

Example: blocks [7/8]

Remark: a **plan** is a sequence of transition leading the initial state to an accepting state

Idea:

- assert property p : “goal state is not reachable”
- if a plan **exists**, NUXMV produces a counterexample for p
- the counterexample for p is a plan to reach the goal

Example: blocks [7/8]

Remark: a **plan** is a sequence of transition leading the initial state to an accepting state

Idea:

- assert property p : “goal state is not reachable”
- if a plan **exists**, NUXMV produces a counterexample for p
- the counterexample for p is a plan to reach the goal

Examples:

- get a plan for reaching “goal state”

SPEC

```
!EF(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 [7/8]

Remark: a **plan** is a sequence of transition leading the initial state to an accepting state

Idea:

- assert property p : “goal state is not reachable”
- if a plan **exists**, NUXMV produces a counterexample for p
- the counterexample for p is a plan to reach the goal

Examples:

- get a plan for reaching “goal state”

SPEC

```
!EF(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

SPEC

```
!EF(block_a.below = none & block_b.below = none &  
    block_c.below = none)
```


Example: blocks [8/8]

- 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`

Example: blocks [8/8]

- 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`

Example: blocks [8/8]

- 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`

- we can always reach a configuration in which all nodes are placed on the table

SPEC

`AG EF (block_a.below = none & block_b.below = none &
block_c.below = none)`

Example: blocks [8/8]

- 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`

- we can always reach a configuration in which all nodes are placed on the table

SPEC

`AG EF (block_a.below = none & block_b.below = none &
block_c.below = none)`

- we can always reach the goal state

SPEC

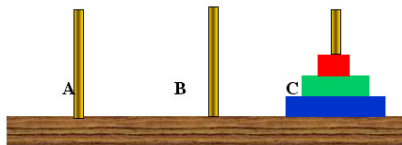
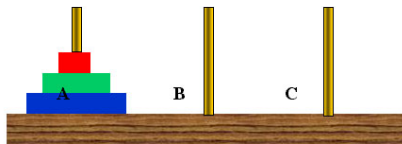
`AG EF(block_a.below = none & block_a.above = b &
block_b.below = a & block_b.above = c &
block_c.below = b & block_c.above = none)`

- 1 Planning problem
 - Blocks Example
- 2 Examples
 - The Tower of Hanoi
 - Ferryman
 - Tic-Tac-Toe
- 3 Exercises

Example: tower of hanoi [1/4]

Game with 3 poles and N disks of different sizes:

- **initial state:** stack of disks with decreasing size on pole A
- **goal state:** move stack on pole C
- **rules:**
 - only one disk may be moved at each transition
 - only the upper disk can be moved
 - a disk can not be placed on top of a smaller disk



Example: tower of hanoi [2/4]

- base system model

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

Example: tower of hanoi [2/4]

- base system model

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

- disk i is moving

```
DEFINE
  move_d1 := (move = 1);
  move_d2 := (move = 2);
  move_d3 := (move = 3);
  move_d4 := (move = 4);
  ...
```


Example: tower of hanoi [2/4]

- base system model

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

- disk i is moving

```
DEFINE
  move_d1 := (move = 1);
  move_d2 := (move = 2);
  move_d3 := (move = 3);
  move_d4 := (move = 4);
  ...
```

- disk d_i can move iff $\forall j > i. d_j \neq d_i$

```
clear_d1 := (d1!=d2 & d1!=d3 & d1!=d4);
clear_d2 := (d2!=d3 & d2!=d4);
clear_d3 := (d3!=d4);
clear_d4 := TRUE;
```

Example: tower of hanoi [3/4]

- initial state

INIT

d1 = left &

d2 = left &

d3 = left &

d4 = left;

Example: tower of hanoi [3/4]

- initial state

```
INIT
```

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

- move description for disk 1

```
TRANS
```

```
move_d1 ->  
  -- disks location changes  
  next(d1) != d1 &  
  next(d2) = d2 &  
  next(d3) = d3 &  
  next(d4) = d4 &  
  -- d1 can move only if it is clear  
  clear_d1 &  
  -- d1 can not move on top of smaller disks  
  next(d1) != d2 &  
  next(d1) != d3 &  
  next(d1) != d4
```

Example: tower of hanoi [4/4]

- get a plan for reaching “goal state”

SPEC

! EF (d1=right & d2=right & d3=right & d4=right)

Example: tower of hanoi [4/4]

- get a plan for reaching “goal state”

SPEC

```
! EF (d1=right & d2=right & d3=right & d4=right)
```

- NUXMV execution:

```
nuXmv > read_model -i hanoi.smv
```

```
nuXmv > go
```

```
nuXmv > check_ctlspec
```

```
-- specification !(EF (((d1 = right & d2 = right) & d3 = right)
                        & d4 = right)) is false
```

```
-- as demonstrated by the following execution sequence
```

```
Trace Description: CTL Counterexample
```

```
-> State: 2.1 <-
```

```
  d1 = left
```

```
  d2 = left
```

```
  d3 = left
```

```
  d4 = left
```

```
...
```

- 1 Planning problem
 - Blocks Example
- 2 Examples
 - The Tower of Hanoi
 - **Ferryman**
 - Tic-Tac-Toe
- 3 Exercises

Example: ferryman [1/4]

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

- **initial state:** all animals are on the right side
- **goal state:** all animals are on the left side
- **rules:**
 - the ferryman can cross the river with at most one passenger on his boat
 - the cabbage and the sheep can not be left unattended on the same side of the river
 - the sheep and the wolf can not be left unattended on the same side of the river

Q: can the ferryman transport all the goods to the other side safely?

Example: 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);
```


Example: 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

```
ASSIGN
  init(cabbage) := right;
  init(sheep)   := right;
  init(wolf)    := right;
  init(man)     := right;
```

Example: ferryman [3/4]

- ferryman carries cabbage

TRANS

```
carry_cabbage ->  
  cabbage = man &  
  next(cabbage) != cabbage &  
  next(man) != man &  
  next(sheep) = sheep &  
  next(wolf) = wolf
```

Example: ferryman [3/4]

- ferryman carries cabbage

TRANS

```
carry_cabbage ->
  cabbage = man &
  next(cabbage) != cabbage &
  next(man) != man &
  next(sheep) = sheep &
  next(wolf) = wolf
```

- ferryman carries sheep

TRANS

```
carry_sheep ->
  sheep = man &
  next(sheep) != sheep &
  next(man) != man &
  next(cabbage) = cabbage &
  next(wolf) = wolf
```

Example: ferryman [3/4]

- ferryman carries cabbage

TRANS

```
carry_cabbage ->
  cabbage = man &
  next(cabbage) != cabbage &
  next(man) != man &
  next(sheep) = sheep &
  next(wolf) = wolf
```

- ferryman carries wolf

TRANS

```
carry_wolf ->
  wolf = man &
  next(wolf) != wolf &
  next(man) != man &
  next(sheep) = sheep &
  next(cabbage) = cabbage
```

- ferryman carries sheep

TRANS

```
carry_sheep ->
  sheep = man &
  next(sheep) != sheep &
  next(man) != man &
  next(cabbage) = cabbage &
  next(wolf) = wolf
```

Example: ferryman [3/4]

- ferryman carries cabbage

TRANS

```
carry_cabbage ->
  cabbage = man &
  next(cabbage) != cabbage &
  next(man) != man &
  next(sheep) = sheep &
  next(wolf) = wolf
```

- ferryman carries sheep

TRANS

```
carry_sheep ->
  sheep = man &
  next(sheep) != sheep &
  next(man) != man &
  next(cabbage) = cabbage &
  next(wolf) = wolf
```

- ferryman carries wolf

TRANS

```
carry_wolf ->
  wolf = man &
  next(wolf) != wolf &
  next(man) != man &
  next(sheep) = sheep &
  next(cabbage) = cabbage
```

- ferryman carries nothing

TRANS

```
no_carry ->
  next(man) != man &
  next(sheep) = sheep &
  next(cabbage) = cabbage &
  next(wolf) = wolf
```

Example: ferryman [4/4]

- get a plan for reaching “goal state”

DEFINE

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

SPEC

```
! E[safe_state U goal]
```

Example: ferryman [4/4]

- get a plan for reaching “goal state”

```
DEFINE
```

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

```
SPEC
```

```
  ! E[safe_state U goal]
```

- NUXMV execution:

```
nuXmv > read_model -i ferryman.smv
```

```
nuXmv > go
```

```
nuXmv > check_ctlspec
```

```
-- specification !E [ safe_state U goal ] is false  
-- as demonstrated by the following execution sequence
```

```
-> State: 1.1 <-  
  cabbage = right  
  sheep = right  
  wolf = right  
  man = right
```

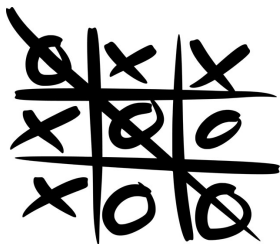
```
...
```

- 1 Planning problem
 - Blocks Example
- 2 Examples
 - The Tower of Hanoi
 - Ferryman
 - Tic-Tac-Toe
- 3 Exercises

Example: tic-tac-toe [1/5]

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

- **Example:** O wins



- we model tic-tac-toe puzzle as an array of size nine

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

Example: tic-tac-toe [2/5]

- base system model

```
MODULE main
```

```
VAR
```

```
  B : array 1..9 of {0,1,2};
```

```
  player : 1..2;
```

```
  move : 0..9;
```

Example: tic-tac-toe [2/5]

- base system model

```
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;
INIT
  move = 0;
```

- turns modeling

ASSIGN

```
init(player) := 1;
next(player) :=
  case
    player = 1 : 2;
    player = 2 : 1;
  esac;
```

Example: tic-tac-toe [3/5]

- turns modeling

ASSIGN

```
init(player) := 1;
next(player) :=
  case
    player = 1 : 2;
    player = 2 : 1;
  esac;
```

- move modeling

TRANS

```
next(move=1) ->
  B[1] = 0 & 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]
```

Example: 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;
```

TRANS

```
(win1 | win2 | draw) <-> next(move)=0
```

Example: tic-tac-toe [5/5]

A **strategy** is a plan that need to be accomplished for winning the game
“if the opponent has two in a row, play the third to block them”

- player 2 does not have a “winning” strategy

SPEC

```
! (AX (EX (AX (EX (AX (EX (AX (EX (AX win2))))))))))
```

- player 2 has a “non-losing” strategy

SPEC

```
AX (EX (AX (EX (AX (EX (AX (EX (AX !win1))))))))
```

Verification:

```
nuXmv > read_model -i tictactoe.smv
```

```
nuXmv > go
```

```
nuXmv > check_ctlspec
```

```
-- specification !(AX (EX (AX (EX (AX (EX  
                (AX (EX (AX win2)))))))))) is true
```

```
-- specification AX (EX (AX (EX (AX (EX  
                (AX (EX (AX !win1)))))))) is true
```

- 1 Planning problem
 - Blocks Example
- 2 Examples
 - The Tower of Hanoi
 - Ferryman
 - Tic-Tac-Toe
- 3 Exercises

- **Tower of Hanoi:** extend the tower of hanoi to handle five disks, and check that the goal state is reachable.
- **Ferryman:** another ferryman has to bring a fox, a chicken, a caterpillar and a crop of lettuce safely across a river.
 - **initial state:** all goods are on the right side
 - **goal state:** all goods 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 of the river
 - the caterpillar eats the lettuce if left unattended on the same side of the river

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

- **Tic-Tac-Toe:** encode and verify the following properties
 - player 2 has also a "non-winning" strategy
 - player 2 does not have a "losing" strategy
 - player 2 does not have a "drawing" strategy
 - player 2 has a "non-drawing" strategy
 - player 1 does not have a "winning" strategy
 - player 1 has a "non-losing" strategy
 - player 1 has also a "non-winning" strategy
 - player 1 does not have a "losing" strategy
 - player 1 does not have a "drawing" strategy
 - player 1 has a "non-drawing" strategy

- will be uploaded on course website within a couple of days
- send me an email if you need help or you just want to propose your own solution for a review

- learning programming languages requires practice: try to come up with your own solutions first!