



UNIVERSITÀ  
DI TRENTO

Department of  
Information Engineering and Computer Science

# Automated Reasoning and **Formal Verification**

## Laboratory 7

Gabriele Masina

[gabriele.masina@unitn.it](mailto:gabriele.masina@unitn.it)

<https://github.com/masinag/arfv2025>

Università di Trento

April 23, 2025



# Outline

---

1. Introduction
2. nuXmv interactive shell
3. nuXmv Modeling
4. Modules
5. Homework



## SMV

Symbolic Model Verifier developed by McMillan in 1993.

## NuSMV

Open-source symbolic model checker for SMV models. It has been developed by FBK, Carnegie Mellon University, the University of Genoa, and the University of Trento.

## nuXmv

Extends NuSMV for infinite state and timed (since v2) systems.  
The binary is available for non-commercial or academic purposes only.<sup>a</sup>  
Developed and maintained by the Formal Methods unit at FBK.

---

<sup>a</sup><https://nuxmv.fbk.eu/download.html>



# Application of nuXmv

---

- ▶ nuXmv allows for the **verification** of:
  - ▶ **finite-state systems** using SAT and BDD based algorithms
  - ▶ **infinite-state systems** (i.e., with *real* and *integer* variables) using SMT-based techniques running on top of MathSAT5
  - ▶ **timed systems** (i.e., with *clock* type) via reduction to infinite state model-checking.
- ▶ nuXmv supports **synchronous composition** of systems
- ▶ Asynchronous composition is no longer supported!



# Outline

---

1. Introduction
2. nuXmv interactive shell
3. nuXmv Modeling
4. Modules
5. Homework



# Interactive shell [1/3]

---

`nuXmv -int` (or `NuSMV -int`) activates an interactive shell

`read_model -i filename` reads the model from the input file.

`go`, `go_bmc`, `go_msat` initialize nuXmv for verification or simulation with a specific backend engine.

`help` shows the list of all commands

`help <command>` shows detailed information for that command

`<command> -h` shows the command line help



## Interactive shell [2/3]

---

`pick_state [-v] [-r | -i [-a]]` picks a state from the set of initial states.

- `-v` prints the chosen state.
- `-r` randomly picks a state from the set of initial states.
- `-i` picks a state from the set of the initial states interactively.
- `-a` displays all state variables (requires `-i`).

`simulate [-p | -v] [-r | -i [-a]] -k N` generates a sequence of at most N transitions starting from the current state.

- `-p` prints the changing variables in the generated trace;
- `-v` prints changed and unchanged variables in the generated trace;
- `-r` at every step picks the next state randomly.
- `-i` at every step picks the next state interactively.
- `-a` prints all state variables (requires `-i`);

`print_current_state [-v]` prints out the current state.

- `-v` prints all the variables.



# Interacting Shell [3/3]

---

`show_vars` `[-s]` `[-f]` `[-i]` `[-t]` `[-v]` prints the variables content and type

- `-s` print state variables;
- `-f` print frozen variables;
- `-i` print input variables;
- `-t` prints the number of variables;
- `-v` prints verbosely;

`reset` resets the whole system (so you can read another model to analyze).

`quit` stops the program.





# Interacting Shell - Output Example

---

```
nuXmv > read_model -i example01.smv ; go
nuXmv > pick_state -v -r
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
b0 = FALSE
b1 = FALSE
nuXmv > simulate -v -r -k 2
***** Simulation Starting From State 1.1
*****
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
b0 = FALSE
b1 = FALSE
-> State: 1.2 <-
b0 = TRUE
b1 = FALSE
-> State: 1.3 <-
b0 = FALSE
b1 = TRUE
```



# Interacting Shell - Output Example

```
nuXmv > read_model -i example01.smv ; go
nuXmv > pick_state -v -r
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
b0 = FALSE
b1 = FALSE
nuXmv > simulate -v -r -k 2
***** Simulation Starting From State 1.1
*****
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
b0 = FALSE
b1 = FALSE
-> State: 1.2 <-
b0 = TRUE
b1 = FALSE
-> State: 1.3 <-
b0 = FALSE
b1 = TRUE
```

## Note

States are numbered as  
`trace_number.state_number`



# Outline

---

1. Introduction

2. nuXmv interactive shell

3. nuXmv Modeling

- Basic Types

- Expressions

- Transition Relation

- Miscellany

- Constraint Style Modeling

4. Modules

5. Homework

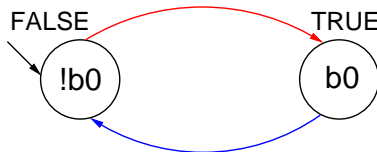


# First SMV model

- ▶ An SMV model is composed of a number of **modules**;
- ▶ Each **module** can contain:
  - ▶ **State variables** declarations;
  - ▶ Formulas defining the valid **initial states**;
  - ▶ Formulas defining the **transition relation**;

## Example

```
MODULE main
VAR
  b0 : boolean;
ASSIGN
  init(b0) := FALSE;
  next(b0) := !b0;
```



Boolean : TRUE, FALSE

x : boolean;

Enumerative :

s : {ready, busy, waiting, stopped};

Bounded integers : (within C/C++ INT\_MIN and INT\_MAX)

n : 1..8;

**integers** : -1, 0, 1, ... (within C/C++ INT\_MIN and INT\_MAX)

n : `integer`;

**rationals** : 1.66, f'2/3, 2e3, 10e-1, ...

r : `real`;

**words** : arrays of bits supporting bitwise logical and arithmetic operations.

unsigned `word`[3];

signed `word`[7];

**arrays** : declared with a pair of lower/upper bounds for the index and a type.  
Array indexes *must be constants*.

**VAR**

```
x : array 0..10 of boolean; -- array of 11 items
y : array -1..1 of {red, green, orange}; -- array of 3 items
z : array 1..10 of array 1..5 of boolean; -- array of arrays
```

**ASSIGN**

```
init(x[5]) := bool(1);
init(y[0]) := {red, green}; -- any value in the set
init(z[3][2]) := TRUE;
```

# Adding a state variable

MODULE main

VAR

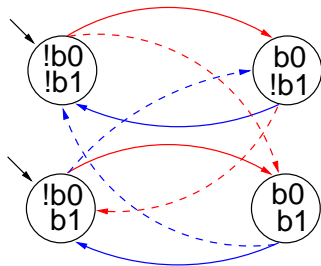
b0 : boolean;

b1 : boolean;

ASSIGN

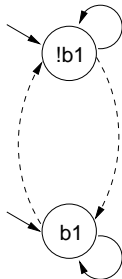
init(b0) := FALSE;

next(b0) := !b0;



Remarks:

- ▶ the FSM is the result of the **synchronous** composition of the “subsystems” for b0 and b1
- ▶ the new state space is the Cartesian product of variables’ ranges.





## Example

```
init(x) := FALSE;      -- x must be FALSE
init(y) := {1, 2, 3};  -- y can be either 1, 2 or 3
```

```
init(<variable>) := <simple_expression>;
```

- ▶ constrains the **initial value** of <variable> to satisfy the <simple\_expression>
- ▶ the initial value of an unconstrained variable can be any of those in its domain

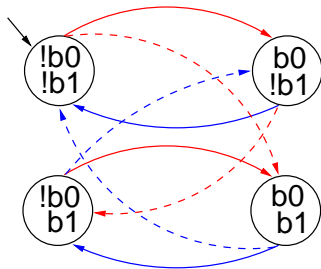
The set of **initial states** is given by the set of states whose variables satisfy *all* the `init(...)` constraints in a module.

## Example

```

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;
ASSIGN
  init(b0) := FALSE;
  next(b0) := !b0;
  init(b1) := FALSE;

```



arithmetic operators :

+      -      \*      /      mod      - (unary)

comparison operators :

=      !=      >      <      <=      >=

logic operators :

&      |      xor      ! (not)      ->      <->

bitwise operators :

<<      >>

set operators : {v1,v2,...,vn}

in : tests a value for membership in a set (*set inclusion*)

union : takes the union of 2 sets (*set union*)

count operator : counts number of true *Boolean* expressions

count(b1, b2, ..., bn)



# Expressions [2/3]

case expression :

```
case
  c1    : e1;
  c2    : e2;
  ...
  TRUE  : en;
esac
```

C/C++ equivalent:

```
if (c1) return e1;
else if (c2) return e2;
...
else return en;
```

if-then-else expression :

```
cond_expr ? basic_expr1 : basic_expr2
```

conversion operators : toint, bool, floor, and

swconst/uwconst : convert an integer to a signed/unsigned word.

word1 : convert boolean to a single word bit.

unsigned/signed : convert signed to unsigned word and vice-versa.

- ▶ Expressions in SMV do not necessarily evaluate to one value. In general, they can represent a set of possible values.  
`init(var) := {a,b,c} union {x,y,z};`
- ▶ The meaning of `:=` in assignments is that the lhs can **non-deterministically** be assigned to any value in the set of values represented by the rhs.
- ▶ A constant `c` is a syntactic abbreviation for `{c}` (the singleton containing `c`).

## Transition Relation

It specifies a constraint on the values that a variable can assume in the **next state**, given the value of variables in the **current state**.

`next(<variable>) := <next_expression>;`

- ▶ `<next_expression>` can depend both on “current” and “next” variables:

`next(a) := { a, a+1 };`

`next(b) := b + (next(a) - a);`

- ▶ `<next_expression>` must evaluate to values in the domain of `<variable>`;
- ▶ the **next** value of an **unconstrained** variable evolves **non-deterministically**;

## Example: Modulo-4 Counter

```
MODULE main
```

```
VAR
```

```
  b0 : boolean;
```

```
  b1 : boolean;
```

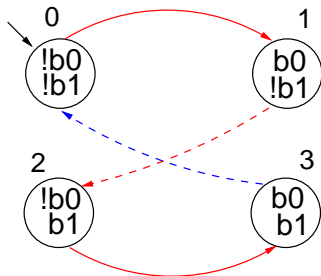
```
ASSIGN
```

```
  init(b0) := FALSE;
```

```
  init(b1) := FALSE;
```

```
  next(b0) := !b0;
```

```
  next(b1) := case
    b0      : !b1;
    TRUE    : b1;
  esac;
```



## Output Variable

A variable whose value deterministically depends on the value of other “current” state variables and for which no `init()` or `next()` are defined.

```
<variable> := <simple_expression>;
```

- ▶ `<simple_expression>` must evaluate to values in the domain of the `<variable>`.
- ▶ used to model *outputs* of a system;



## Example: Modulo-4 Counter + Output

```
MODULE main
```

```
VAR
```

```
  b0 : boolean;
```

```
  b1 : boolean;
```

```
  out : 0..3;
```

```
ASSIGN
```

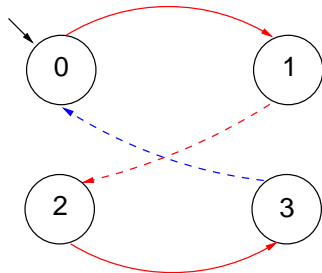
```
  init(b0) := FALSE;
```

```
  next(b0) := !b0;
```

```
  init(b1) := FALSE;
```

```
  next(b1) := ((!b0 & b1) | (b0 & !b1));
```

```
  out      := toint(b0) + 2 * toint(b1);
```





# Assignment Rules ( $:=$ )

Single assignment rule :Each variable may be **assigned only once**. **Illegal** examples.

<code>init(var) := ready;</code>	<code>var := ready;</code>	<code>next(var) := ready;</code>
<code>init(var) := busy;</code>	<code>var := busy;</code>	<code>var := busy;</code>
<code>next(var) := ready;</code>	<code>init(var) := ready;</code>	
<code>next(var) := busy;</code>	<code>var := busy;</code>	



# Assignment Rules ( $:=$ )

**Single assignment rule** :Each variable may be **assigned only once**. **Illegal** examples.

<code>init(var) := ready;</code>	<code>var := ready;</code>	<code>next(var) := ready;</code>
<code>init(var) := busy;</code>	<code>var := busy;</code>	<code>var := busy;</code>

<code>next(var) := ready;</code>	<code>init(var) := ready;</code>
<code>next(var) := busy;</code>	<code>var := busy;</code>

**Circular dependency rule** :A set of equations must not form *cycles* in their dependency graph, unless broken by delays. **Illegal** examples:

<code>next(x) := next(y);</code>	<code>x := (x + 1) mod 2;</code>	<code>next(x) := x &amp; next(x);</code>
<code>next(y) := next(x);</code>		

Instead, the following is **legal**:

<code>next(x) := next(y);</code>
<code>next(y) := y &amp; x;</code>

# DEFINE declarations

**DEFINE** <id> := <simple\_expression>;

- ▶ Each occurrence of the defined symbol is replaced with the body of the definition
- ▶ Alternative way to define *output variables*;

## Example

```
MODULE main
```

```
VAR
```

```
  b0 : boolean;
```

```
  b1 : boolean;
```

```
ASSIGN
```

```
  init(b0) := FALSE; next(b0) := !b0;
```

```
  init(b1) := FALSE; next(b1) := ((!b0 & b1) | (b0 & !b1));
```

```
DEFINE
```

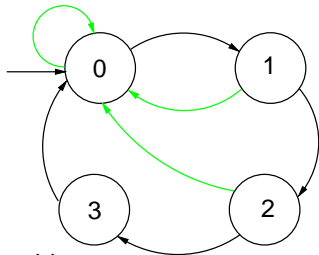
```
  out := toint(b0) + 2 * toint(b1);
```

# Example: Modulo-4 Counter with Reset

The counter can be reset by an external “uncontrollable” signal.

```

MODULE main
VAR
  b0 : boolean; b1 : boolean; reset : boolean;
ASSIGN
  init(b0) := FALSE; init(b1) := FALSE;
  next(b0) := case reset : FALSE;
                    TRUE  : !b0;
                esac;
  next(b1) := case reset : FALSE;
                    TRUE  : ((!b0 & b1) | (b0 & !b1));
                esac;
DEFINE out := toint(b0) + 2 * toint(b1);
  
```



## Excercise 7.1

Simulate the system with nuXmv and draw the FSM.

```
MODULE main
```

```
VAR
```

```
    request : boolean;
```

```
    state   : { ready, busy };
```

```
ASSIGN
```

```
    init(state) := ready;
```

```
    next(state) :=
```

```
        case
```

```
            state = ready & request : busy;
```

```
            TRUE                     : { ready, busy };
```

```
        esac;
```

## Excercise 7.1

Simulate the system with nuXmv and draw the FSM.

```
MODULE main
```

```
VAR
```

```
    request : boolean;
```

```
    state   : { ready, busy };
```

```
ASSIGN
```

```
    init(state) := ready;
```

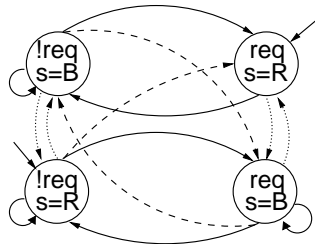
```
    next(state) :=
```

```
        case
```

```
            state = ready & request : busy;
```

```
            TRUE                     : { ready, busy };
```

```
        esac;
```





# Constraint Style Modeling [1/4]

---

- ▶ Up to now, we have seen how to define a model in **assignment style**:

```
MODULE main
VAR request : boolean;  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case state = ready & request : busy;
                  TRUE                        : {ready,busy};
                  esac;
```

- ▶ Every program can be alternatively defined in a **constraint style**:

```
INIT
  state = ready
TRANS
  (state = ready & request) -> next(state) = busy
```



- ▶ A model can be specified by zero or more **constraints** on:
  - ▶ **initial states:**  
`INIT` <simple\_expression>
  - ▶ **transitions:**  
`TRANS` <next\_expression>
  - ▶ **invariants:**  
`INVAR` <simple\_expression>
- ▶ Any propositional or SMT formula can be used as constraint;
- ▶ Constraints can be **mixed** with assignments;
- ▶ **Not all constraints can be easily rewritten in terms of assignments!**

`TRANS`

$$\text{next}(b0) + 2 * \text{next}(b1) + 4 * \text{next}(b2) = \\ (b0 + 2 * b1 + 4 * b2 + \text{tick}) \bmod 8$$



## Assignment Style

- ▶ By construction, there is always **at least one initial state**
- ▶ By construction, all states have **at least one next state**
- ▶ **Non-determinism is apparent** (unassigned variables, set assignments. . .)

## Constraint Style

- ▶ **INIT** constraints **can be inconsistent**  $\Rightarrow$  **no initial state!**
  - ▶ Any specification (also **SPEC** 0) is vacuously true.
- ▶ **TRANS** constraints **can be inconsistent**  $\Rightarrow$  **deadlock state!**

```
MODULE main
```

```
VAR b : boolean;
```

```
TRANS b -> FALSE;
```

**tip:** use `check_fsm` to detect deadlock states

- ▶ **non-determinism is hidden**

```
TRANS (state = ready & request) -> next(state) = busy
```



# Example: Constraint Style & Case

```
MODULE main
```

```
VAR
```

```
    state : {S0, S1, S2};
```

```
DEFINE
```

```
    go_s1 := state != S2;
```

```
    go_s2 := state != S1;
```

```
INIT
```

```
    state = S0;
```

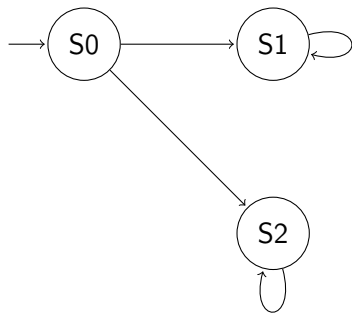
```
TRANS
```

```
case
```

```
    go_s1 : next(state) = S1;
```

```
    go_s2 : next(state) = S2;
```

```
esac;
```



Q: does it correspond to the FSM?



# Example: Constraint Style & Case

```
MODULE main
```

```
VAR
```

```
  state : {S0, S1, S2};
```

```
DEFINE
```

```
  go_s1 := state != S2;
```

```
  go_s2 := state != S1;
```

```
INIT
```

```
  state = S0;
```

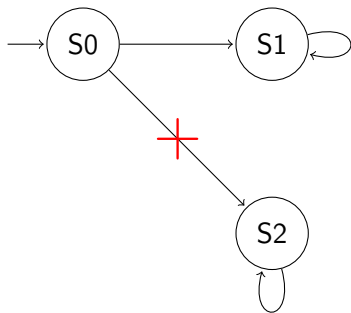
```
TRANS
```

```
case
```

```
  go_s1 : next(state) = S1;
```

```
  go_s2 : next(state) = S2;
```

```
esac;
```



Q: does it correspond to the FSM? No: cases are evaluated in order!

# Example: Constraint Style & Swap

```
MODULE main
```

```
VAR
```

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

```
  i : 0..1;
```

```
ASSIGN
```

```
  init(arr[0]) := 1;
```

```
  init(arr[1]) := 2;
```

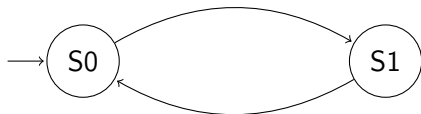
```
  init(i) := 0;
```

```
  next(i) := 1-i;
```

```
TRANS
```

```
  next(arr[i]) = arr[1-i] &
```

```
  next(arr[1-i]) = arr[i];
```



arr[0] = 1

arr[1] = 2

i = 0

arr[0] = 2

arr[1] = 1

i = 1

Q: does it correspond to the FSM?

# Example: Constraint Style & Swap

MODULE main

VAR

arr: array 0..1 of {1,2};

i : 0..1;

ASSIGN

init(arr[0]) := 1;

init(arr[1]) := 2;

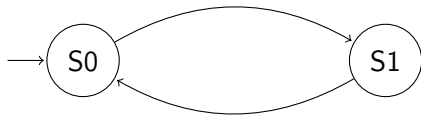
init(i) := 0;

next(i) := 1-i;

TRANS

next(arr[i]) = arr[1-i] &

next(arr[1-i]) = arr[i];



arr[0] = 1

arr[1] = 2

i = 0

arr[0] = 1

arr[1] = 2

i = 1

Q: **does it correspond to the FSM?** No: everything inside the `next()` operator is evaluated within the next state, indexes included!



# Outline

---

1. Introduction

2. nuXmv interactive shell

3. nuXmv Modeling

4. Modules

Modules Definition

Modules Composition

5. Homework



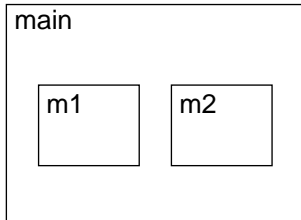
A nuXmv program is composed of a main module plus 0 or *more* other modules:

- ▶ a module can be **instantiated** as a VAR in other modules
- ▶ variables **local** to a module instance are accessed via **dot notation** (e.g., m1.out).

## Example

```
MODULE counter
VAR out: 0..9;
ASSIGN next(out) := (out + 1) mod 10;
```

```
MODULE main
VAR m1 : counter; m2 : counter;
    sum : 0..18;
ASSIGN sum := m1.out + m2.out;
```



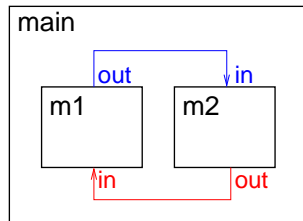
A module declaration can be *parametric*:

- ▶ a parameter is passed *by reference*;
- ▶ any expression can be used as parameter;

## Example

```
MODULE counter(in)
VAR out: 0..9;
...

MODULE main
VAR m1 : counter(m2.out);
    m2 : counter(m1.out);
...
```



- ▶ Modules can be **composed**
- ▶ Modules *without parameters and assignments* can be seen as simple **records**

## Example

```
MODULE point
```

```
VAR
```

```
x: -10..10;
```

```
y: -10..10;
```

```
MODULE circle
```

```
VAR
```

```
center: point;
```

```
radius: 0..10;
```

```
MODULE main
```

```
VAR c: circle;
```

```
ASSIGN
```

```
init(c.center.x) := 0;
```

```
init(c.center.y) := 0;
```

```
init(c.radius)   := 5;
```

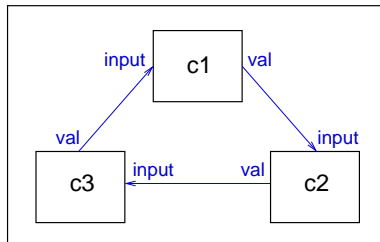
# Synchronous composition [1/2]

The composition of modules is **synchronous**: *all modules move at each step.*

```

MODULE cell(input)
VAR
  val : {red, green, blue};
ASSIGN
  next(val) := input;

MODULE main
VAR
  c1 : cell(c3.val);
  c2 : cell(c1.val);
  c3 : cell(c2.val);
  
```



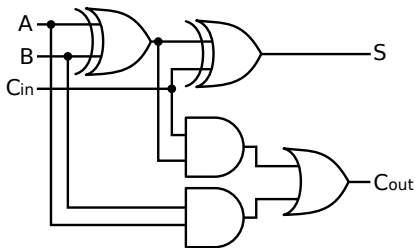
A possible execution:

step	c1.val	c2.val	c3.val
0	red	green	blue
1	blue	red	green
2	green	blue	red
3	red	green	blue
4	...	...	...
5	red	green	blue

# Exercise: Adder [1/3]

## Exercise 7.2: Binary Adder

Implement a binary adder that takes into account two 4-bits numbers and returns their sum using an output variable. Implement both a bit-adder and the general adder as two separate modules.



Licensed under [CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/). Source: <https://commons.wikimedia.org/wiki/File:Full-adder.svg>



## Exercise: Adder [2/3]

---

```
MODULE bit-adder(in1, in2, cin)
VAR sum : boolean; cout : boolean;
ASSIGN next(sum) := (in1 xor in2) xor cin;
      next(cout) := (in1 & in2) | ((in1 xor in2) & cin);
```

```
MODULE adder(in1, in2)
VAR bit[0] : bit-adder(in1[0], in2[0], bool(0));
    bit[1] : bit-adder(in1[1], in2[1], bit[0].cout);
    bit[2] : bit-adder(in1[2], in2[2], bit[1].cout);
    bit[3] : bit-adder(in1[3], in2[3], bit[2].cout);
DEFINE sum[0] := bit[0].sum;
    sum[1] := bit[1].sum;
    sum[2] := bit[2].sum;
    sum[3] := bit[3].sum;
    overflow := bit[3].cout;
```

## Exercise: Adder [3/3]

---

```
MODULE main
VAR in1 : array 0..3 of boolean;
    in2 : array 0..3 of boolean;
    a   : adder(in1, in2);
ASSIGN next(in1[0]) := in1[0]; next(in1[1]) := in1[1];
      next(in1[2]) := in1[2]; next(in1[3]) := in1[3];
      next(in2[0]) := in2[0]; next(in2[1]) := in2[1];
      next(in2[2]) := in2[2]; next(in2[3]) := in2[3];
DEFINE op1 := toint(in1[0]) + 2*toint(in1[1]) + 4*toint(in1[2]) +
      8*toint(in1[3]);
      op2 := toint(in2[0]) + 2*toint(in2[1]) + 4*toint(in2[2]) +
      8*toint(in2[3]);
      sum := toint(a.sum[0]) + 2*toint(a.sum[1]) + 4*toint(a.sum[2]) +
      8*toint(a.sum[3]) + 16*toint(a.overflow);
```





# Outline

---

1. Introduction
2. nuXmv interactive shell
3. nuXmv Modeling
4. Modules
5. Homework

## Homework 7.1: playing with Adder

- ▶ Simulate a random execution of the “Adder” system;
- ▶ After how many steps the adder stores the computed final `sum` value? Is this number constant? Can you explain its behaviour?
- ▶ What happens if we initialize both `sum` and `cout` inside the bit-adder to `FALSE`? Can you tell the main difference with respect to the original algorithm?
- ▶ Can you modify the model in a simple way so that the `sum` is obtained after a single iteration? (PS: simple means you must modify/add less than 5 lines of code)
- ▶ Add a `reset` control which changes the values of the operands and restarts the computation of the `sum`

## Homework 7.2: Random Calculator

Use nuXmv to create a “random” calculator:

- ▶ it creates two random arrays of 3 integers numbers in the range  $[1,10]$
- ▶ it randomly chooses an operator to apply to each pair of items in the arrays (sum, subtraction and multiplication), storing the result in an output array of 3 elements called `res`
- ▶ the results must be defined in 3 steps:
  - ▶ in the first iteration you'll store the random operation between elements with index 0
  - ▶ in the second iteration the random operation between elements with index 1
  - ▶ and the same for the last index

Use an additional variable, `index`, to take into account this evolution.