Université Nice Sophia Antipolis

NuSMV

Hands-on introduction

F. Mallet

fmallet@unice.fr

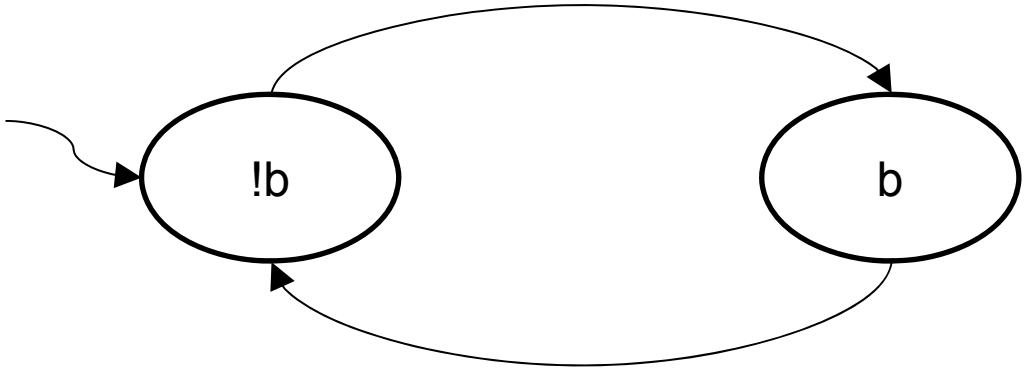
Université Nice Sophia Antipolis

NuSMV 2

- NuSMV 1 was an extension of SMV
  - SMV : first BDD-based symbolic model-checker [McMillan, 90]
- NuSMV 2
  - Combines BDD-based and SAT-based model-checking
  - OpenSource licensing (GNU LGPL)

Kripke Structure

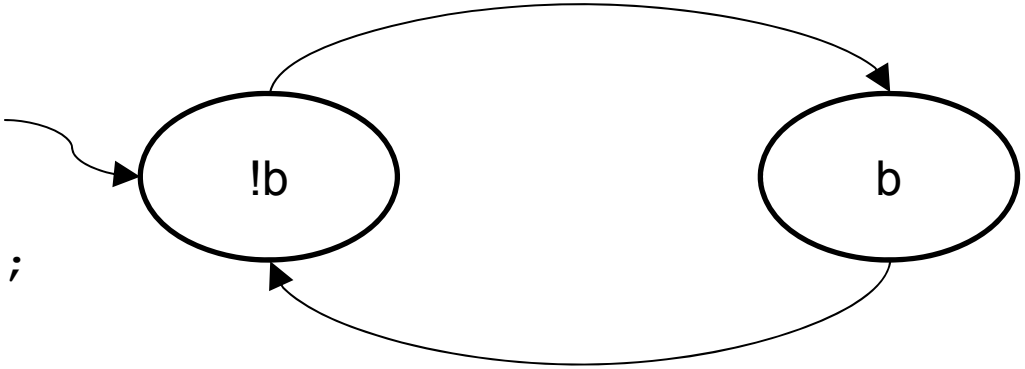
- Non-deterministic automaton used in model-checking (graph)
  - **[Nodes]** Finite reachable states of the system (S)
    - Set of initial states $I \subseteq S$
  - **[Edges]** state transition $R \subseteq S \times S$
  - Labeling function L
    - Maps each node to a set of properties that hold in the state



First NuSMV program

- (Parameterized) Modules
  - Declarations of state variables
    - The state variables determine the state space
    - The domain must be bounded (and small)
  - Assignments
    - Valid initial states : init
    - Transition relation : next
- Example

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



Data types (1/2)

- Boolean
  - VAR

    b : boolean; — FALSE, TRUE

- Integer
  - VAR

    i : integer; — Signed 32-bit

    j : 1..8;

- Enumeration types
  - VAR

    thread : { stopped, running, waiting, finished }; — Symbolic constants

    t1 : {2, 4, -2, 0}; — Integer enum

    t2 : { FAIL, 1, 3, 7, OK }; — Integer-and-symbolic

Data types (2/2)

- Unsigned words (vectors of bits/boolean)
  - VAR

    b : unsigned word [3]; Unsigned 3-bit

- Signed words (vectors of bits/boolean)
  - VAR

    i : signed word [7]; Signed 7-bit

- Array
  - VAR

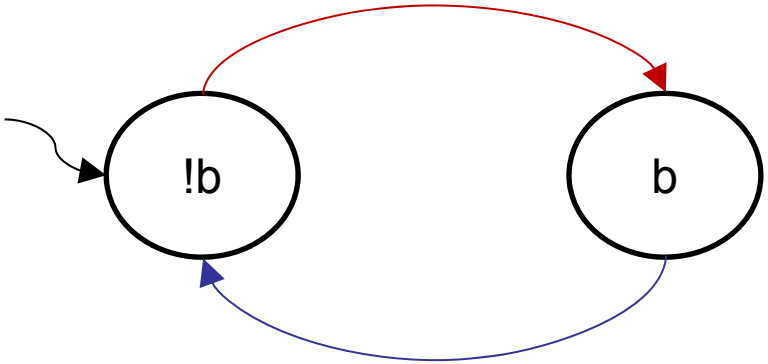
    a : array 0..3 of boolean;

    a1 : array 10..20 of {OK, y, z};

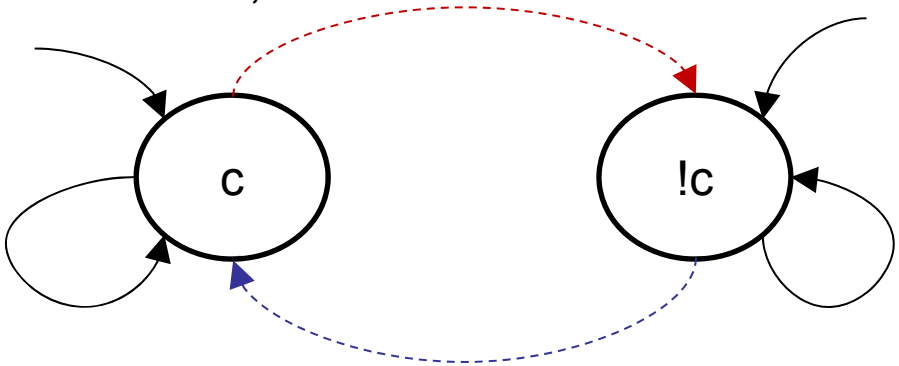
    a2 : array 1..8 of array -1..2 of unsigned word[5];

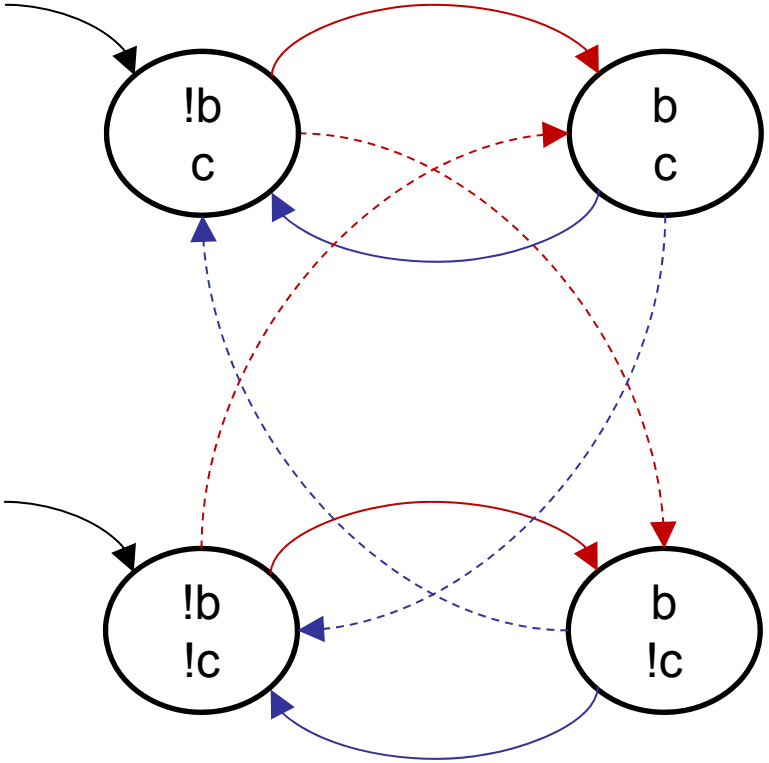
Synchronous composition

- All the transitions evolve synchronously



c : boolean ;

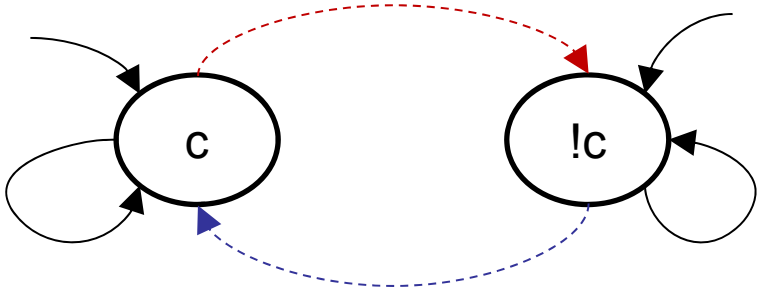




Composed state-space=
Cartesian product

Initial states: **init**

- If an initial value is not specified
  - The variable can initially assume any value in its domain
    - var c : boolean;

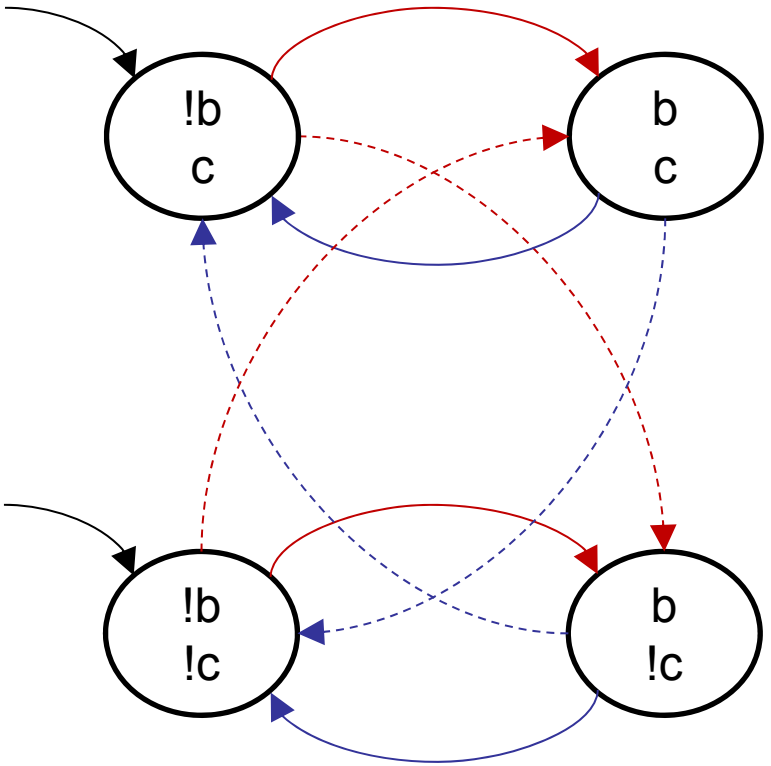


- Otherwise, an expression can compute (a set of) values in the definition domain
  - init(b) := FALSE ; ⇔ init(b) := { FALSE } ;
  - init(c) := { TRUE, FALSE } ; (non-deterministic)
  - init(var) := {a,b,c} union {x,y,z} ;

Initial states: **init**

- Non-deterministic initialization

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



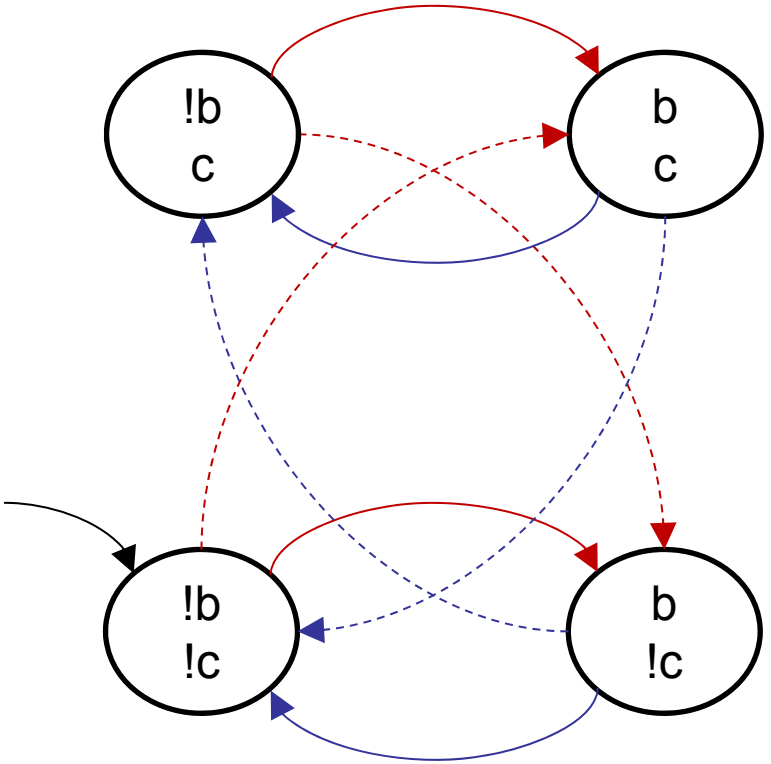
Composed state-space=
Cartesian product

Initial states: **init**

- Deterministic initialization

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

    init(c) := FALSE;
```



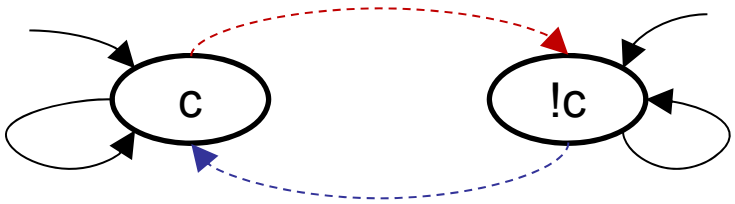
Composed state-space=
Cartesian product

Transition relation: **next**

- Constrain the values that a variable can assume in the next state
  - next(<variable>) := <expression>;
- <expression> depends on the definition domain of <variable>
  - The expression can depends on current and next values of variables.
    - next(output) := ! input;
    - next(output) := (!input) **union** output;
    - next(b) := b + (next(a) – a);

Transition relation: **next**

- If no next assignment
  - The variable evolves non-deterministically

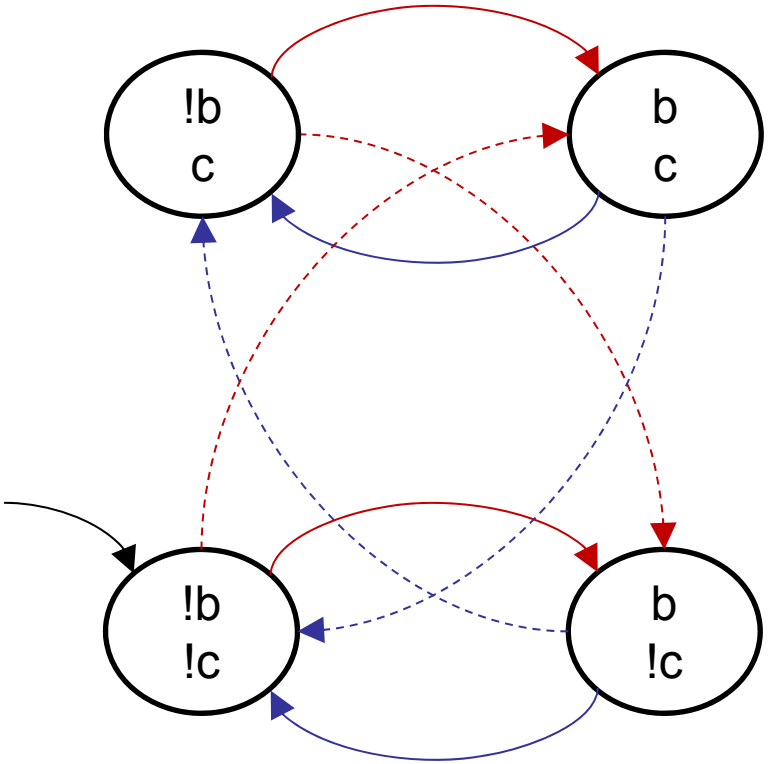


Transition relation: **next**

- Non-deterministic transition

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

    init(c) := FALSE;
```



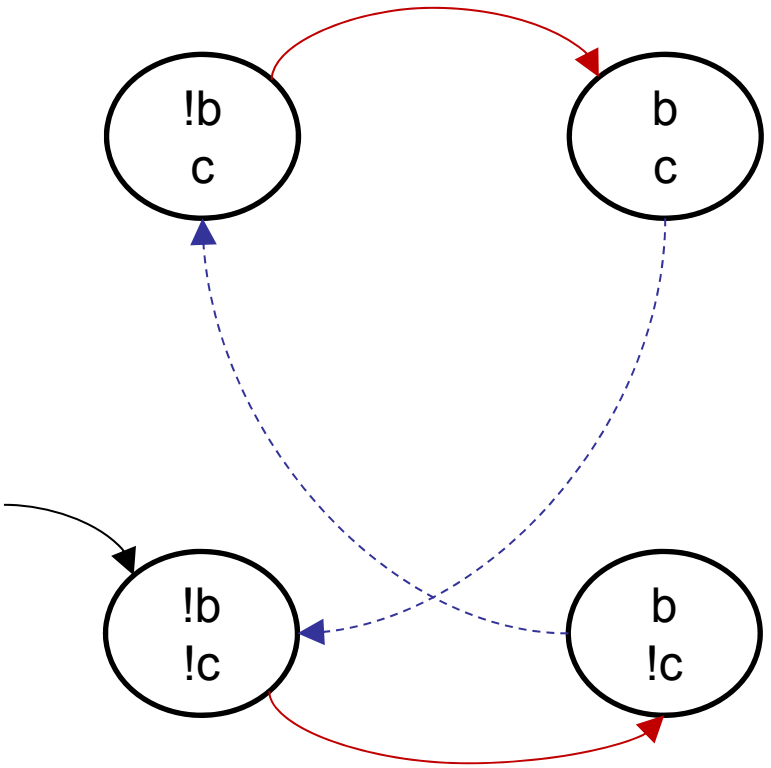
Composed state-space=
Cartesian product

Transition relation: **next**

- Deterministic transition

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

    init(c) := FALSE;
    next(c) := (!b & c) |
               (b & !c) ;
```



Composed state-space=
Cartesian product

Normal assignments

- When neither init nor next is used
  - Specify the current value of a variable depending on the current values of (other) variables
    - <variable> := <expression> ;

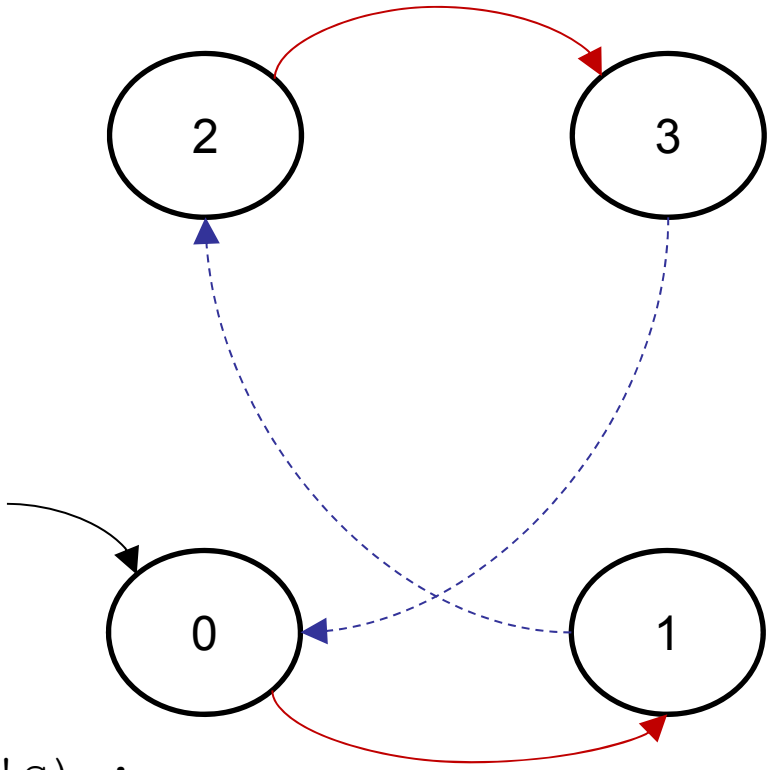
Normal assignments

Counter

```
MODULE main
  VAR
    b : boolean;
    c : boolean;
    out : 0..3;
  ASSIGN
    init(b) := FALSE;
    next(b) := !b;

    init(c) := FALSE;
    next(c) := (!b & c) | (b & !c) ;

    out := b + 2*c;
```

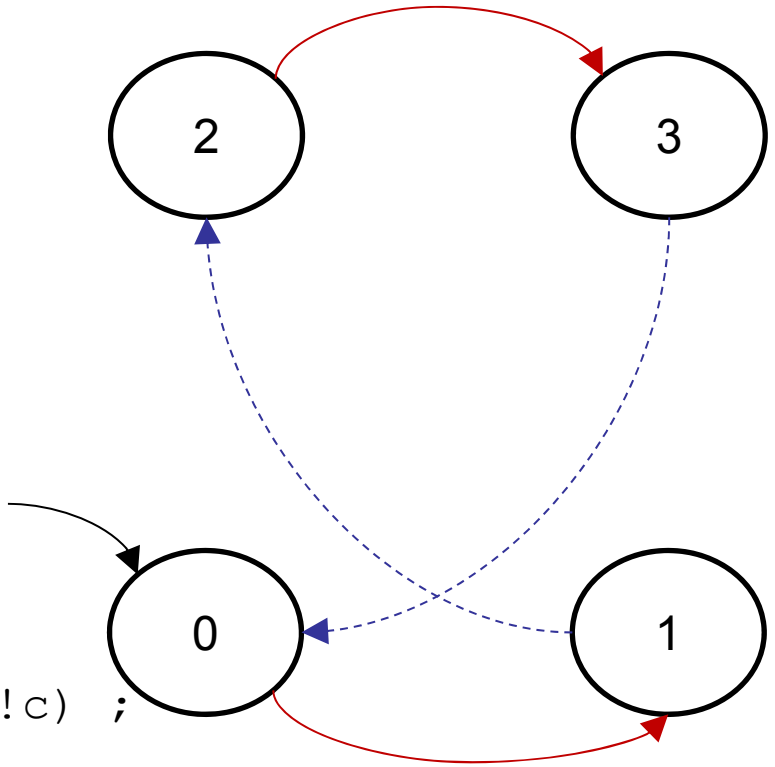


Definitions

Counter

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

    init(c) := FALSE;
    next(c) := (!b & c) | (b & !c) ;
DEFINE
    out := b + 2*c;
```



Definitions are NOT statically typed !

Restrictions

❑ Double-assignment rule

- Each variable may be assigned only once in the program
- Example of illegal assignments
  - init(status) := ready;
  - status := busy;

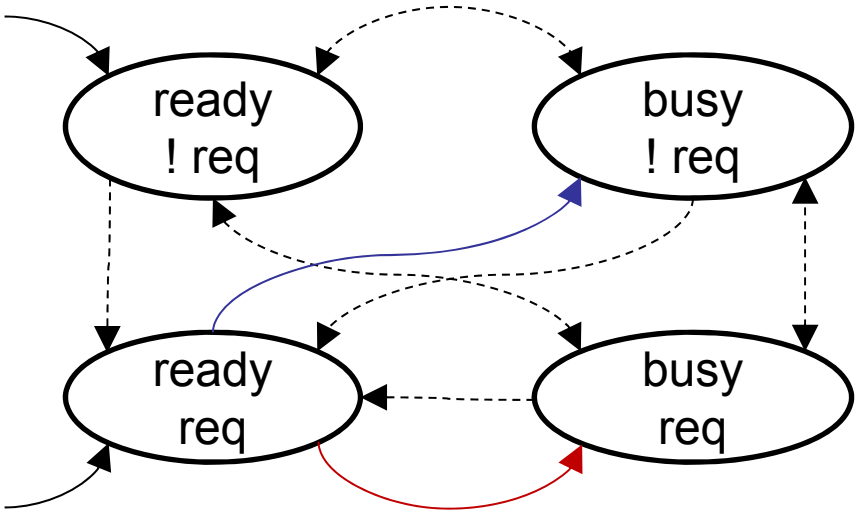
❑ Circular-dependency rule

- A variable cannot have cycles in its dependency graph not broken by delays
  - x := (x+1) ;
  - next(x) := x & next(y);
  - next(y) := y & next(x);

Simple example

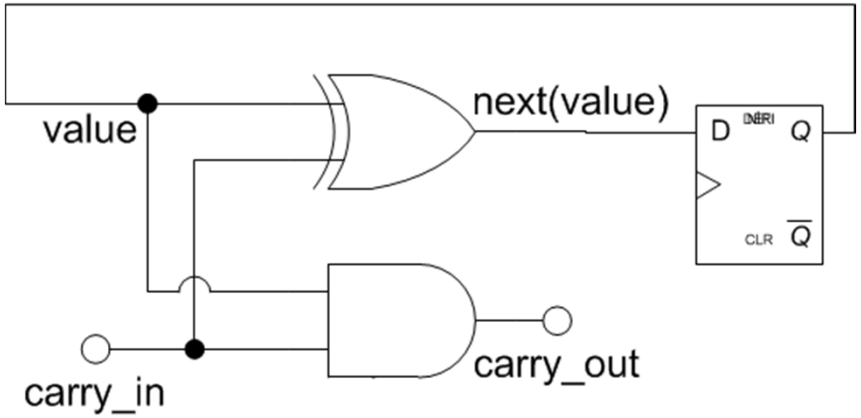
- Case expressions
  - Keep the first condition that is evaluated to TRUE
  - At least one condition must be TRUE

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



Module with parameters

- 1-bit counter



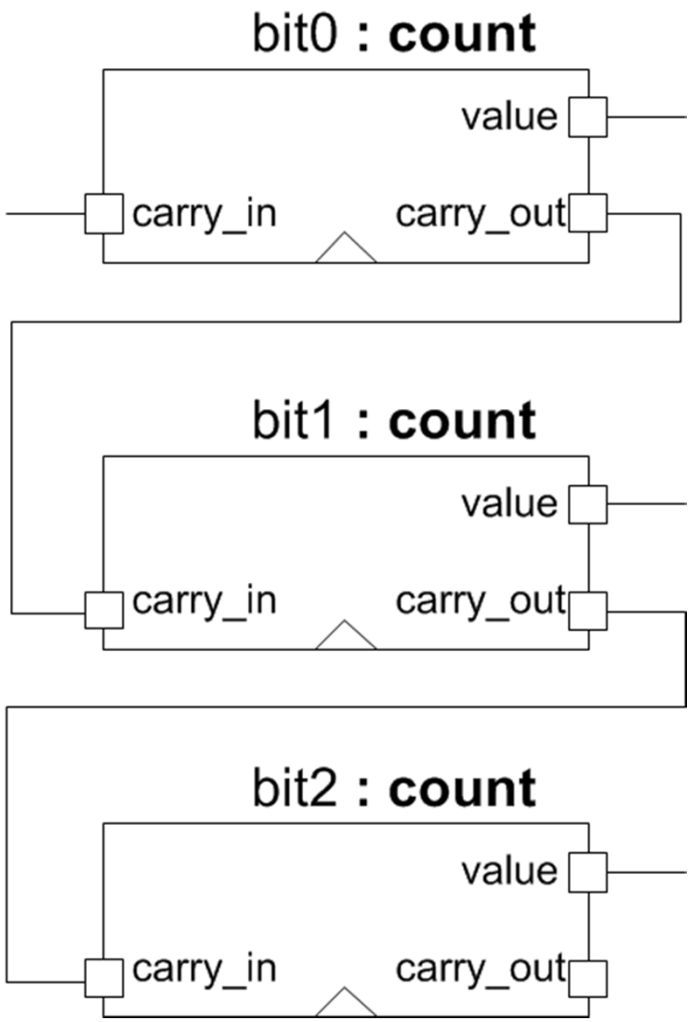

```
MODULE count(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in
```

Module with parameters

- 3-bit counter

```
MODULE main
VAR
  bit0 : count(TRUE);
  bit1 : count(bit0.carry_out);
  bit2 : count(bit1.carry_out);

MODULE count(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in
```



Specifications

- Specifications are associated with modules
  - Each property is verified separately
  - Several kinds of specifications
- **Invariants**: `INVARSPEC`
  - Something that must always be true
  - A proposition on reachable states
- **Linear-Time Logics** (LTL) : `LTLSPEC`
- **Branching-Time Logics** (CTL) : `SPEC`

CTL formulas: SPEC (1/2)

```
ctl_expr ::
  simple_expr
  | ( ctl_expr )
  | ! ctl_expr
  | ctl_expr & ctl_expr
  | ctl_expr | ctl_expr
  | ctl_expr xor ctl_expr
  | ctl_expr xnor ctl_expr
  | ctl_expr -> ctl_expr
  | ctl_expr <-> ctl_expr
```

A CTL expression can be

- a simple expression (no next)
- Logical not
- Logical and
- Logical or
- Logical exclusive or
- Logical NOT exclusive or
- Logical implies
- Logical equivalence

CTL formulas: `SPEC` (2/2)

`| EG ctl_expr`	Exists globally
`| EX ctl_expr`	Exists next state
`| EF ctl_expr`	Exists finally
`| AG ctl_expr`	Forall globally
`| AX ctl_expr`	Forall next state
`| AF ctl_expr`	Forall finally
`| E [ ctl_expr U ctl_expr ]`	Exists until
`| A [ ctl_expr U ctl_expr ]`	Forall until

Invariants: `INVARSPEC`

- Equivalent to
  - SPEC AG simple_expr;
- But
  - Checked separately
  - Can contain next operators

LTL formulas: `LTLSPEC` (1/3)

```
ltl_expr ::
  simple_expr
  | ( ctl_expr )
  | ! ctl_expr
  | ctl_expr & ctl_expr
  | ctl_expr | ctl_expr
  | ctl_expr xor ctl_expr
  | ctl_expr xnor ctl_expr
  | ctl_expr -> ctl_expr
  | ctl_expr <-> ctl_expr
```

A LTL expression can be

- a simple expression (no next)
- Logical not
- Logical and
- Logical or
- Logical exclusive or
- Logical NOT exclusive or
- Logical implies
- Logical equivalence

LTL formulas: `LTLSPEC` (2/3)

```
| G ltl_expr
| X ltl_expr
| F ltl_expr
| p U q
```

Future expressions

- globally
- next state
- finally
- Until
  - q must hold at some point (eventually)
  - p must hold all the time until then
  - p and q need not hold at the same time

```
| p V q
```

- Releases
  - p must hold at all time t'>=t up to and including the time step t'' where q also holds
  - q may never hold

LTL formulas: `LTLSPEC` (3/3)

```
| H ltl_expr
| Y ltl_expr
| O ltl_expr
| p S q
```

Past expressions

- historically
- previous state
- once
- Since
  - q must have held at some point (t'<=t)
  - p must hold all the time at t'' (t'<t''<=t)
  - p need not hold at t'

```
| p T q
```

- Triggered
  - p held at t'<=t
  - q holds at all t'', (t'<=t''<=t)
  - Or
    - p never held
    - q must hold at all t'' (t0<=t''<=t)