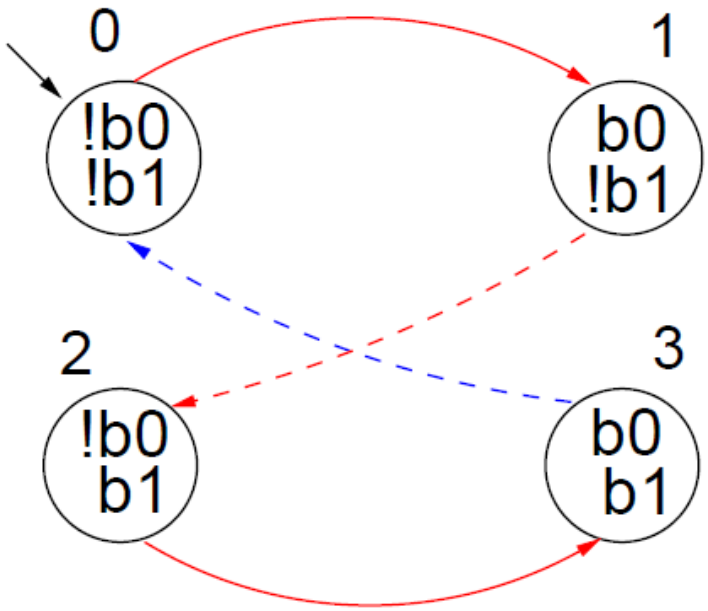
Cours 4 : NuSMV

Nga Nguyen

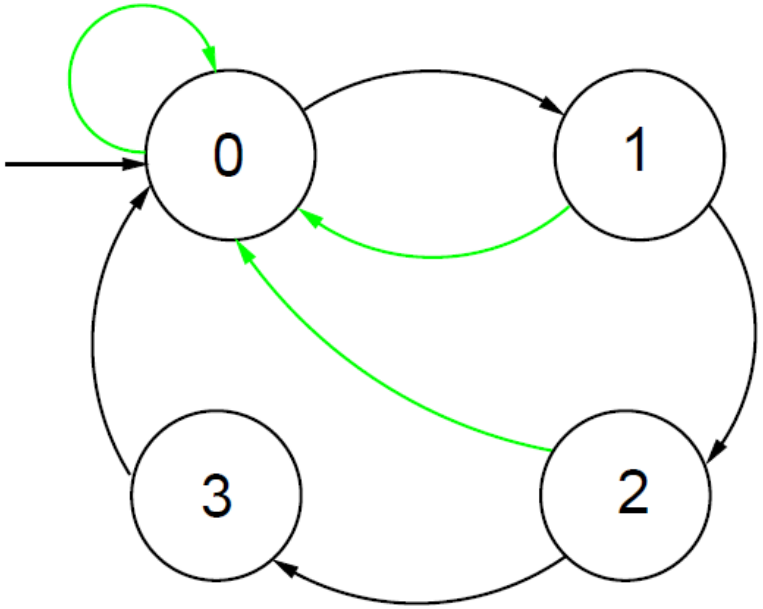
Compteur modulo 4

```
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);
```



Compteur modulo 4

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



Spécifications

- Invariant :
  - INVARSPEC out < 2
- CTLSPEC :
  - Il est possible d'atteindre un état où out = 3 :
  - Il est inévitable que out = 3 est finalement atteignable :
  - Il est toujours possible d'atteindre un état où out = 3 :
  - Chaque fois que l'état où out = 2 est atteint, un état avec out = 3 est atteint après :
  - L'opération reset est correcte :

Spécifications

- Invariant :
  - INVARSPEC out < 2

- CTLSPEC :
  - Il est possible d'atteindre un état où out = 3 :

    EF out = 3
  - Il est inévitable que out = 3 est finalement atteignable :

    AF out = 3
  - Il est toujours possible d'atteindre un état où out = 3 :

    AG EF out = 3
  - Chaque fois que l'état où out = 2 est atteint, un état avec out = 3 est atteint après :

    AG (out = 2 -> AF out = 3)
  - L'opération reset est correcte :

    AG (reset -> AX out = 0)

Spécifications

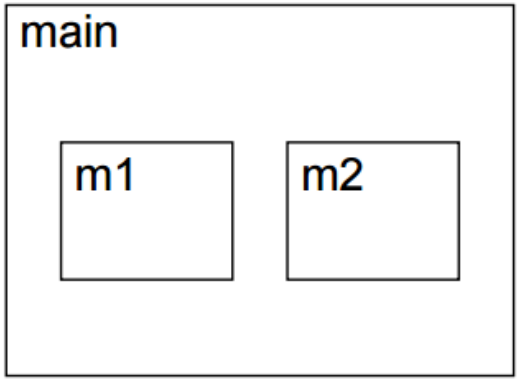
- Est-ce que AF out = 1 est vrai ?
- AF out = 2 ?

- Contrainte d'équité :
  - éliminer les comportements dans lesquels certaines conditions ne tiennent jamais / considérer seulement les exécutions satisfaisant infiniment souvent ces conditions
  - FAIRNESS/JUSTICE out = 3

Modules

- Un programme NuSMV peut avoir plusieurs déclarations de module :

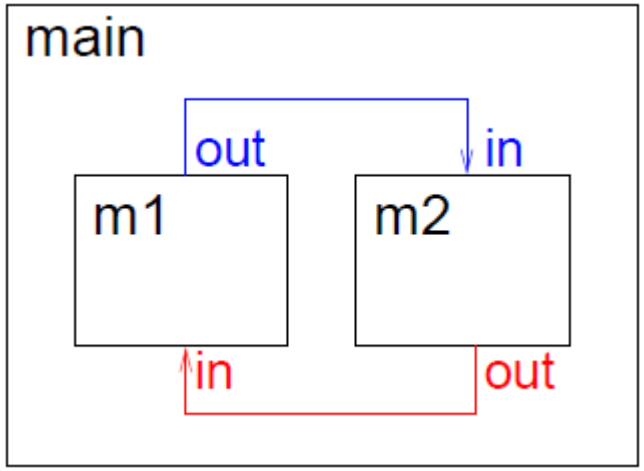
```
MODULE modu
    VAR out: 0..9;
    ASSIGN
            next(out) := (out + 1) mod 10;

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



- Chaque programme doit avoir un module **main**
- Un module peut être “instancié” plusieurs fois, chaque instanciation crée une copie des variables locales

Paramètres de module

```
MODULE modu(in)
    VAR out: 0..9;
    ...
MODULE main
    VAR m1 : modu(m2.out);
         m2 : modu(m1.out);
    ...
```



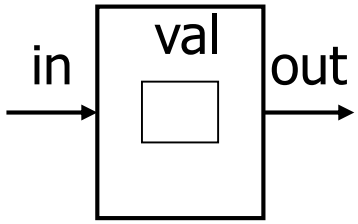
- Passage par référence !

A Three-Bit Counter

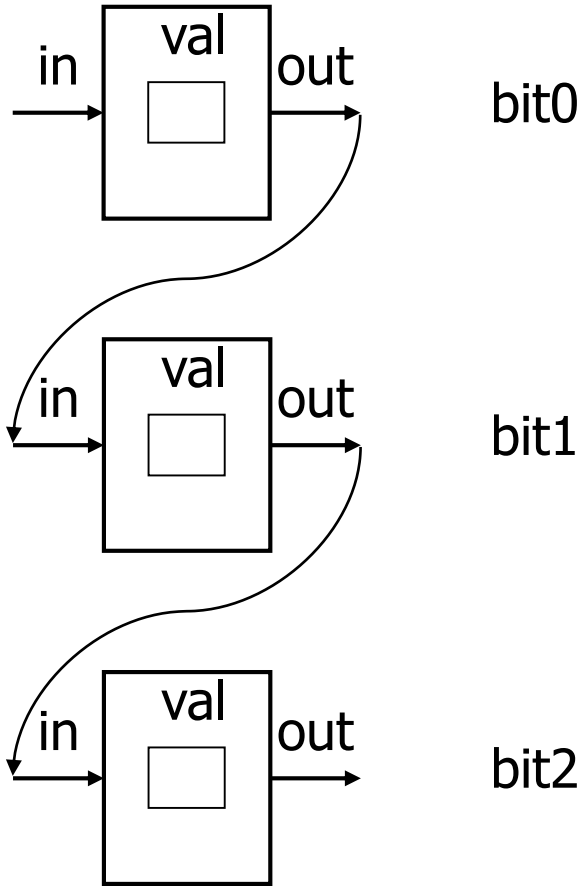
```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;

MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
  res : 0..7;
ASSIGN
  res := toint(bit0.value)+2*toint(bit1.value)+4*toint(bit2.value);

SPEC  AG AF bit2.carry_out
```

(value + carry_in) mod 2

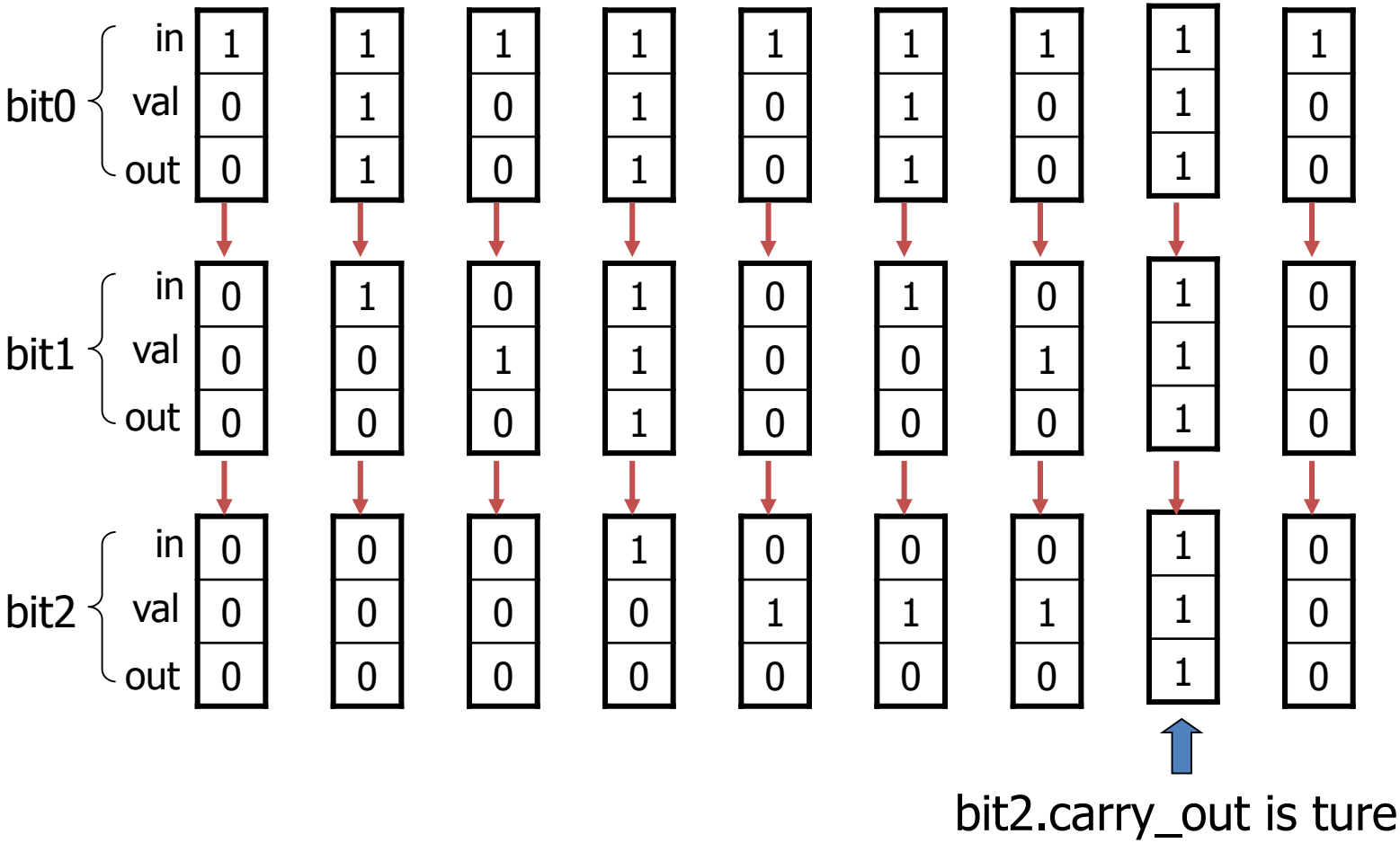
module declaration



module instantiations



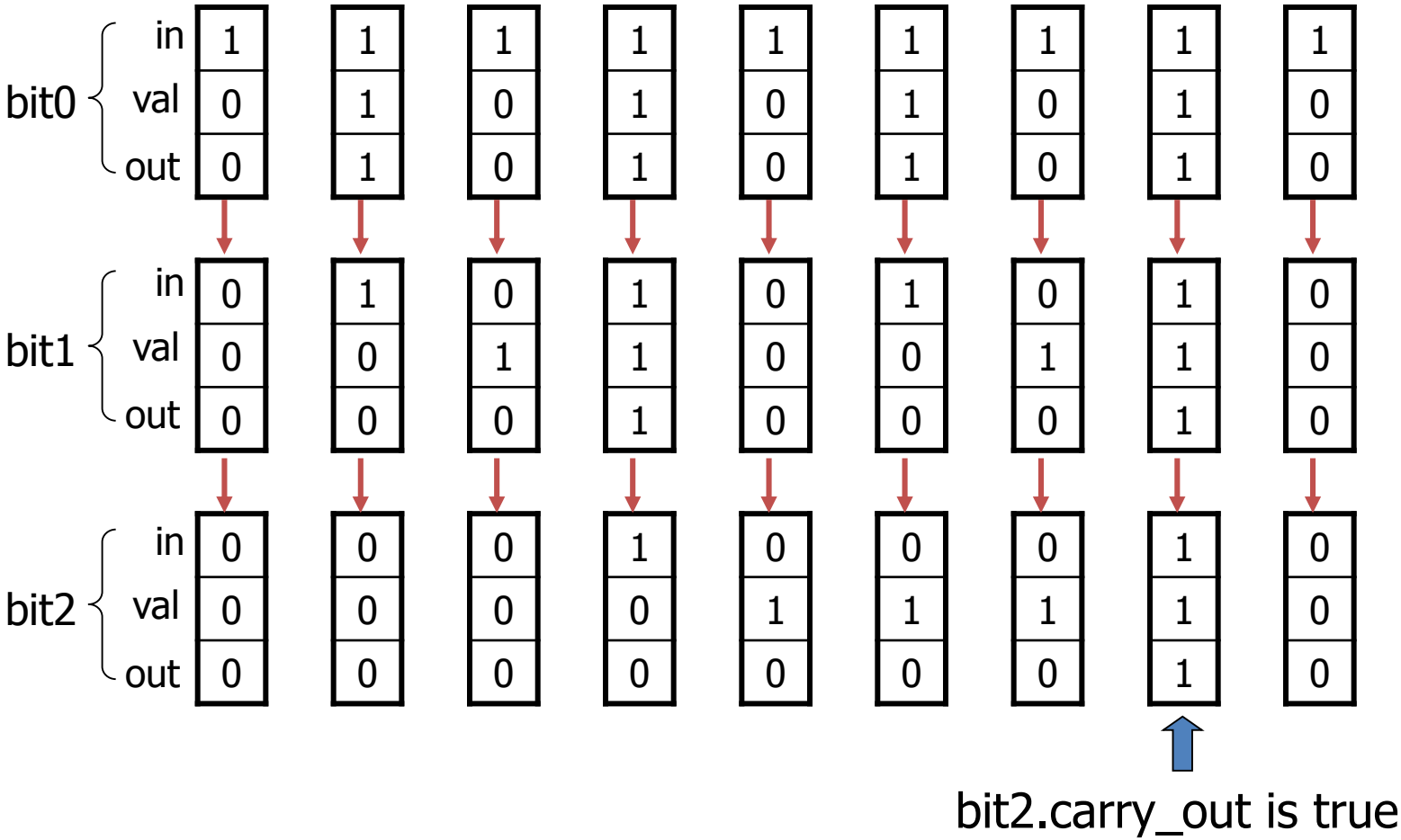
AG AF bit2.carry_out is true



A Three-Bit Counter

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

MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
  res : 0..7;
ASSIGN
  res := toint(bit0.value)+2*toint(bit1.value)+4*toint(bit2.value

SPEC AG (!bit2.carry_out)
```

`AG (!bit2.carry_out)` is false



Composition de module

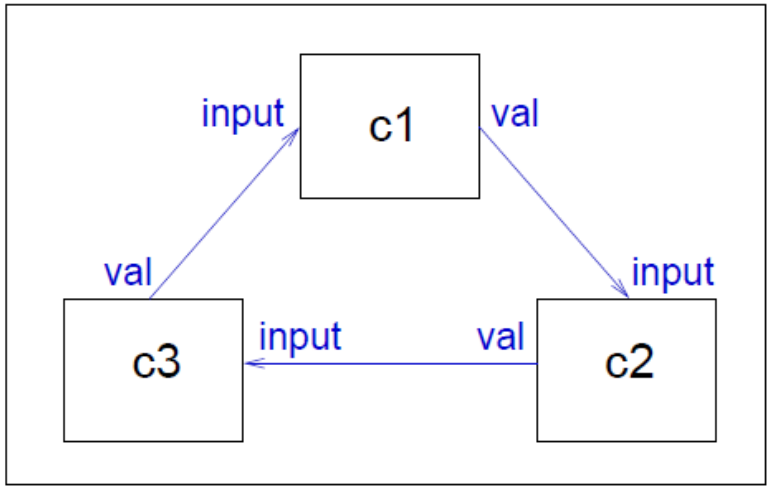
- **Composition synchrone**
  - Les composants évoluent en parallèle
  - A chaque instance, chaque composant réalise sa transition
  - Par défaut

- **Composition asynchrone**
  - Entrelacement entre les composants
  - A chaque instance, un seul composant est choisi pour réaliser sa transition
  - Mot-clé : **process**

Exemple

- Par défaut, la composition est synchrone :

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

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



Une exécution possible

step	*c1.val*	*c2.val*	*c3.val*
0	**red**	**green**	**blue**
1	red	**red**	**green**
2	**green**	red	green
3	green	red	green
4	green	red	**red**
5	**red**	**green**	red
6	red	**red**	red
7	red	red	red
8	red	red	red
9	red	red	red
10	red	red	red

Exemple

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

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

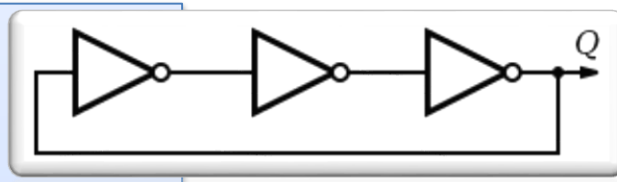
- Une variable booléenne *running* est définie dans chaque processus
  - Elle est vraie quand le processus est choisi
  - Elle peut être utilisée pour garantir l'équité entre les processus

Une exécution possible

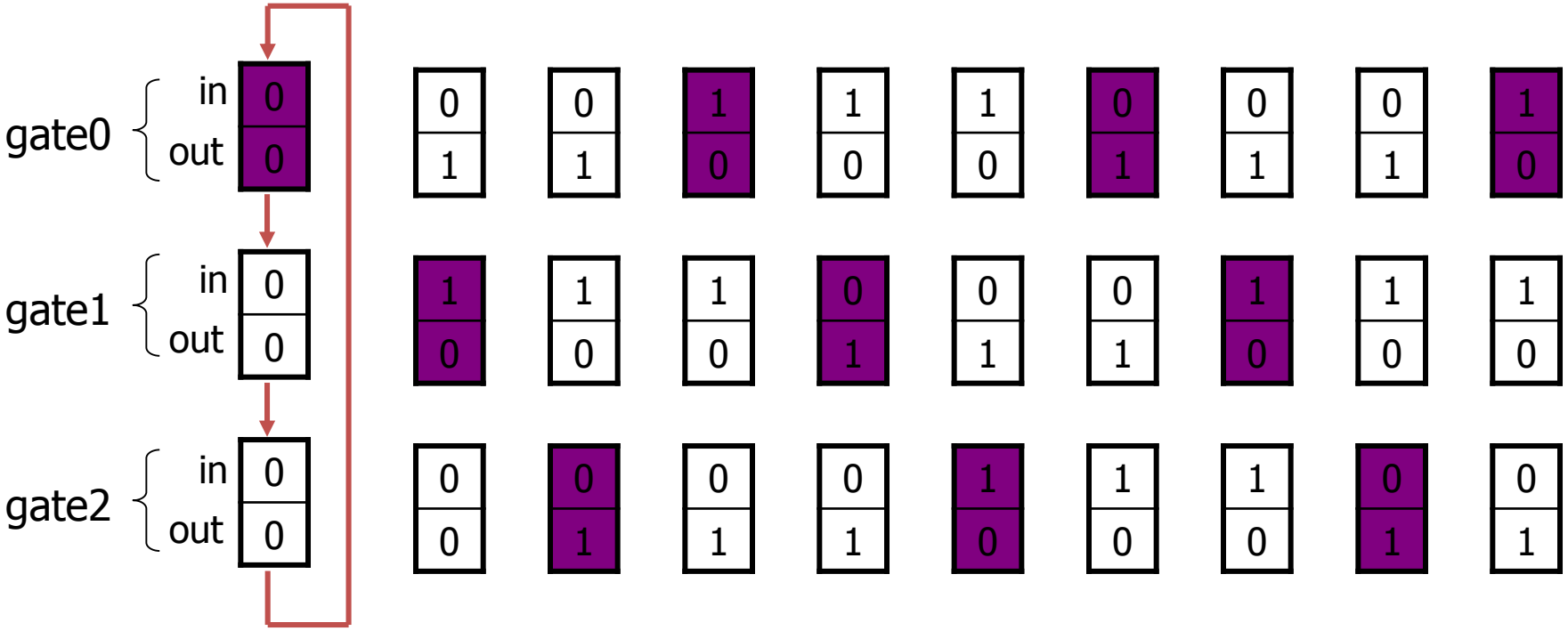
step	*running*	*c1.val*	*c2.val*	*c3.val*
0	-	**red**	**green**	**blue**
1	c2	red	**red**	blue
2	c1	**blue**	red	blue
3	c1	blue	red	blue
4	c2	blue	red	blue
5	c3	blue	red	**red**
6	c2	blue	**blue**	red
7	c1	blue	blue	red
8	c1	**red**	blue	red
9	c3	red	blue	**blue**
10	c3	red	blue	blue

Inverter Ring

```
MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);

SPEC (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := FALSE;
  next(output) := !input;

FAIRNESS
  running
```



In asynchronous composition, a step of the computation is a step by exactly one component. The process to execute is assumed to choose gate0, gate1, and gate2 repeatedly.



`(AG AF gate1.output) & (AG AF !gate1.output)` is true

NuSMV : simulation

- pick_state [-v] [-r | -i] picks a state from the set of initial states
  - -v prints the chosen state
  - -r picks a state from the set of the initial states randomly
  - -i picks a state from the set of the initial states interactively
- simulate [-p | -v] [-r | -i] -k steps generates a sequence of at most steps states starting from the current state.
  - -p prints only the changed state variables.
  - -v prints all the state variables.
  - -r at every step picks the next state randomly.
  - -i at every step picks the next state interactively.

NuSMV : simulation

- goto_state state_label makes state_label the current state
- show_traces [-v] [trace_number] shows the trace identied by trace_number or the most recently generated trace if trace_number is omitted.

  -v prints all the state variables
- print_current_state [-h] [-v] prints out the current state.

  -v prints all the variables