

# NuSMV \*

Alberto Griggio

`alberto.griggio@disi.unitn.it`

`http://disi.unitn.it/~griggio/formal\_methods.html`

Course on Formal Methods

Lab Class April 8, 2008

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Simulation</b>	<b>2</b>
<b>3</b>	<b>Modeling</b>	<b>3</b>
3.1	Basic Definitions . . . . .	3
3.2	Modules . . . . .	9
3.3	Constraint style . . . . .	12
3.4	Synchronous vs. Asynchronous . . . . .	14
<b>4</b>	<b>Examples</b>	<b>16</b>
4.1	Synchronous . . . . .	16
4.2	Asynchronous . . . . .	19

---

\*These slides are derived from those by Stefano Tonetta (Univ of Lugano), used for the FM lab 2005/06

# 1 Introduction

## Introduction

- NuSMV is a symbolic model checker developed by FBK-IRST.
- The NuSMV project aims at the development of a state-of-the-art model checker that:
  - is robust, open and customizable;
  - can be applied in technology transfer projects;
  - can be used as research tool in different domains.
- NuSMV is *OpenSource*:
  - developed by a distributed community,
  - “Free Software” license.
- NuSMV home page:
  - <http://nusmv.fbk.eu/>

# 2 Simulation

## Interactive shell

- “NuSMV `-int`” activates an interactive shell
- “`set input_file filename`” sets the input model.
- “`go`” reads and initializes NuSMV for verification or simulation.
- “`pick_state [-v]`” picks a state from the set of initial state.
  - “`-v`” prints the chosen state.
- “`simulate [-p | -v] [-r | -i] steps`” generates a sequence of at most `steps` states starting from the `current state`.
  - “`-p`” and “`-v`” print the generated trace. “`-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.

## Inspecting traces

- “`goto_state state_label`” makes `state_label` the `current state`.
- “`show_traces [-v] [trace_number]`” shows the trace identified by `trace_number` or the most recently generated trace if `trace_number` is omitted.
  - “`-v`” prints prints all the state variables.

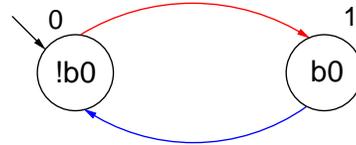
## 3 Modeling

### 3.1 Basic Definitions

The first **SMV** program

```
MODULE main
  VAR
    b0 : boolean;

  ASSIGN
    init(b0) := 0;
    next(b0) := !b0;
```



An SMV program consists of:

- Declarations of the state variables (**b0** in the example); the state variables determine the state space of the model.
- Assignments that define the valid initial states (`init(b0) := 0`).
- Assignments that define the transition relation (`next(b0) := !b0`).

#### Declaring state variables

The SMV language provides booleans, enumerative, bounded integers and words as data types:

```
boolean:   x : boolean;
enumerative: st : {ready, busy, waiting, stopped};
bounded integers (intervals): n : 1..8;
words:    w : word[8];
```

#### Arrays

The SMV language provides also the possibility to define *arrays*.

```
VAR
  x : array 0..10 of boolean;
  y : array 2..4 of 0..10;
  z : array 0..10 of array 0..5 of {red, green, orange};
ASSIGN
  init(x[5]) := 1;
  init(y[2]) := {0,2,4,6,8,10};
  init(z[3][2]) := {green, orange};
```

Remarks:

- Array indexes in SMV *must be constants*;

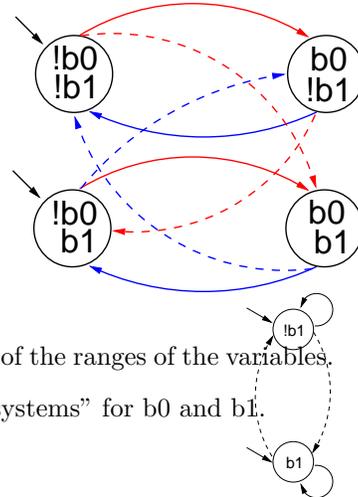
### Adding a state variable

```

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;

```



Remarks:

- The new state space is the cartesian product of the ranges of the variables.
- Synchronous composition between the “subsystems” for b0 and b1.

### Declaring the set of initial states

- For each variable, we constrain the values that it can assume in the *initial states*.

```

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

```

- <simple\_expression> must evaluate to values in the domain of <variable>.
- If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.

### Declaring the set of initial states

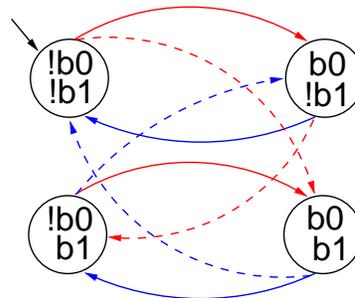
```

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;

  init(b1) := 0;

```



## Expressions

- Arithmetic operators: +   -   \*   /   mod   - (unary)
- Comparison operators: =   !=   >   <   <=   >=
- Logic operators: &   |   xor   ! (not)   ->   <->
- Conditional expression:

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

if c1 then e1 else if c2 then e2 else if ... else en
- Set operators: {v1,v2,...,vn} (enumeration)   in (set inclusion)   union (set union)

## Expressions

- 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 assume non-deterministically a value in the set of values represented by the rhs.
- A constant c is considered as a syntactic abbreviation for {c} (the singleton containing c).

## Declaring the transition relation

- The transition relation is specified by constraining the values that variables can assume in the *next state*.

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

- <next\_expression> must evaluate to values in the domain of <variable>.
- <next\_expression> depends on “current” and “next” variables:

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

- If no next() assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.

Unconstrained variables can be used to model non-deterministic *inputs* to the system.

## Declaring the transition relation

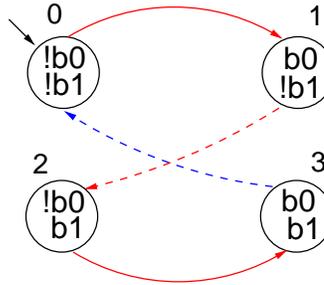
```

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;

  init(b1) := 0;
  next(b1) := ((!b0 & b1) | (b0 & !b1));

```



## Specifying normal assignments

- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

`<variable> := <simple_expression> ;`

- `<simple_expression>` must evaluate to values in the domain of the `<variable>`.

## Specifying normal assignments

```

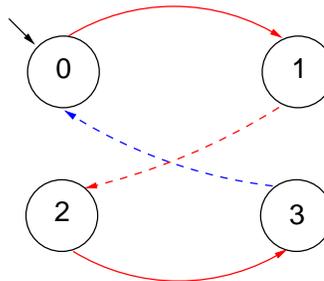
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;
  out : 0..3;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;

  init(b1) := 0;
  next(b1) := ((!b0 & b1) | (b0 & !b1));

  out := b0 + 2*b1;

```



### Restrictions on the ASSIGN

In order for an SMV program to be implementable, assignments have the following restrictions:

- Double assignments rule – Each variable may be assigned only once in the program.
- Circular dependencies rule – A variable cannot have “cycles” in its dependency graph that are not broken by delays.

If an SMV program does not respect these restrictions, an error is reported by NuSMV.

### Double assignments rule

**Each variable may be assigned only once in the program.**

```
init(status) := ready;
init(status) := busy;          ILLEGAL!
```

```
next(status) := ready;
next(status) := busy;          ILLEGAL!
```

```
status := ready;
status := busy;                ILLEGAL!
```

```
init(status) := ready;
status := busy;                ILLEGAL!
```

```
next(status) := ready;
status := busy;                ILLEGAL!
```

### Circular dependencies rule

**A variable cannot have “cycles” in its dependency graph that are not broken by delays.**

```
x := (x + 1) mod 2;            ILLEGAL!
```

```
x := (y + 1) mod 2;
y := (x + 1) mod 2;            ILLEGAL!
```

```
next(x) := x & next(x);        ILLEGAL!
```

```
next(x) := x & next(y);
next(y) := y & next(x);        ILLEGAL!
```

```
next(x) := x & next(y);
next(y) := y & x;              LEGAL!
```

### The DEFINE declaration

- DEFINE declarations can be used to define *abbreviations*.
- An alternative to normal assignments.
- Syntax:

```
DEFINE <id> := <simple_expression> ;
```

- They are similar to macro definitions.[6mm]
- No new state variable is created for defined symbols (hence, no added complexity to model checking).[6mm]
- Each occurrence of a defined symbol is replaced with the body of the definition.

### The DEFINE declaration

```
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;
  init(b1) := 0;
  next(b1) := ((!b0 & b1) | (b0 & !b1));

DEFINE
  out := b0 + 2*b1;
```

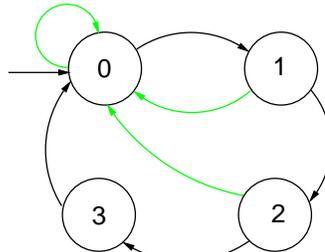
### Example: A modulo 4 counter with reset

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

```

MODULE main
VAR
  b0 : boolean; b1 : boolean; reset : boolean;
ASSIGN
  init(b0) := 0; init(b1) := 0;
  next(b0) := case
    reset = 1 : 0;
    reset = 0 : !b0;
  esac;
  next(b1) := case
    reset : 0;
    1 : ((!b0 & b1) | (b0 & !b1));
  esac;
DEFINE
  out := b0 + 2*b1;

```



### 3.2 Modules

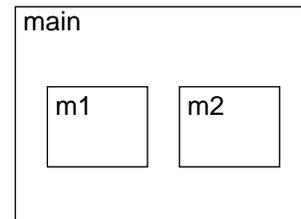
#### Modules

An SMV program can consist of one or more *module declarations*.

```

MODULE mod
  VAR out: 0..9;
  ASSIGN next(out) :=
    (out + 1) mod 10;
MODULE main
  VAR m1 : mod; m2 : mod;
  sum: 0..18;
  ASSIGN sum := m1.out + m2.out;

```



- Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- In each SMV specification there must be a module `main`.
- All the variables declared in a module instance are referred to via the dot notation (e.g., `m1.out`, `m2.out`).

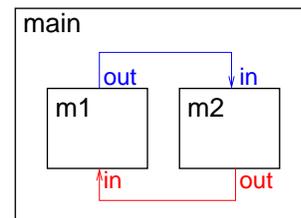
#### Module parameters

Module declarations may be *parametric*.

```

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

```



- *Formal parameters* (*in*) are substituted with the *actual parameters* (*m2.out*, *m1.out*) when the module is instantiated.
- Actual parameters can be any legal expression.
- Actual parameters are passed by reference.

**Example: The modulo 4 counter revisited**

```

MODULE counter_cell(tick)
VAR
    value : boolean;
    done  : boolean;

ASSIGN
    init(value) := 0;
    next(value) := case
        tick = 0 : value;
        tick = 1 : (value + 1) mod 2;
    esac;
    done := tick & (((value + 1) mod 2) = 0);

```

Remarks: *tick* is the formal parameter of module *counter\_cell*.

**Example: The modulo 4 counter revisited**

```

MODULE main
VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.done);
    out  : 0..3;
ASSIGN
    out := bit0.value + 2*bit1.value;

```

Remarks:

- Module *counter\_cell* is instantiated two times.
- In the instance *bit0*, the formal parameter *tick* is replaced with the actual parameter *1*.
- When a module is instantiated, all variables/symbols defined in it are preceded by the module instance name, so that they are unique to the instance.

## Module hierarchies

```
MODULE counter_4(tick)
VAR
  bit0 : counter_cell(tick);
  bit1 : counter_cell(bit0.done);
  out : 0..3; done : boolean;
ASSIGN out:= bit0.value + 2*bit1.value;
DEFINE done := bit1.done;

MODULE counter_64(tick) -- A counter modulo 64
VAR
  b0 : counter_4(tick);
  b1 : counter_4(b0.done);
  b2 : counter_4(b1.done);
  out : 0..63;
ASSIGN out := b0.out + 4*b1.out + 16*b2.out;
```

## The modulo 4 counter with reset revisited

```
MODULE counter_cell(tick, reset)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value):=
    case
      reset = 1 : 0;
      1 : case
        tick = 0 : value;
        tick = 1 : (value + 1) mod 2;
      esac;
    esac;
DEFINE
  done := tick & (((value + 1) mod 2) = 0);
```

## The modulo 4 counter with reset revisited

```
MODULE counter_4(tick, reset)
VAR
  bit0 : counter_cell(tick, reset);
  bit1 : counter_cell(bit0.done, reset);
DEFINE
  out := bit0.value + 2*bit1.value;
  done := bit1.done;
```

```

MODULE main
VAR
  reset : boolean;
  c : counter_4(1, reset);
DEFINE
  out := c.out;

```

### Records

Records can be defined as modules without parameters and assignments.

```

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;

```

## 3.3 Constraint style

### The constraint style of model specification

```

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

```

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

```

MODULE main
VAR request : boolean;
  state : {ready,busy};
INIT state = ready
TRANS (state = ready & request) -> next(state) = busy

```

### The constraint style of model specification

- The SMV language allows for specifying the model by defining constraints on:

- the *states*: INVAR <simple\_expression>
- the *initial states*: INIT <simple\_expression>
- the *transitions*: TRANS <next\_expression>
- There can be zero, one, or more constraints in each module, and constraints can be mixed with assignments.
- Any propositional formula is allowed in constraints.
- INVAR *p* is equivalent to INIT *p* and TRANS next(*p*), but is more efficient.
- Risk of defining *inconsistent models* (INIT *p* & !*p*).

### Assignments versus constraints

- Any ASSIGN-based specification can be easily rewritten as an equivalent constraint-based specification:

```

ASSIGN
init(state) := {ready, busy};   INIT state in {ready, busy}
next(state) := ready;           TRANS next(state) = ready
out := b0 + 2*b1;               INVAR out = b0 + 2*b1

```

- The converse is not true: the following constraint

```

TRANS
next(b0) + 2*next(b1) + 4*next(b2) =
    (b0 + 2*b1 + 4*b2 + tick) mod 8

```

cannot be easily rewritten in terms of ASSIGNS.

### Assignments versus constraints

- Models written in **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...).
- Models written in **constraint style**:
  - INIT constraints *can be inconsistent*:
    - \* inconsistent model: no initial state,
    - \* any specification (also SPEC 0) is vacuously true.
  - TRANS constraints *can be inconsistent*:
    - \* the transition relation is not total (there are deadlock states),

\* NuSMV can detect and report this case (`check_fsm`).

\* Example:

```
MODULE main
  VAR b : boolean;
  TRANS b = 1 -> 0;
```

– *non-determinism is hidden* in the constraints

### The modulo 4 counter with reset, using constraints

```
MODULE counter_cell(tick, reset)
  VAR
    value : boolean;
    done : boolean;
  INIT
    value = 0;
  TRANS
    reset = 1 -> next(value) = 0
  TRANS
    reset = 0 -> ((tick = 0 -> next(value) = value) &
                  (tick = 1 -> next(value) = (value+1) mod 2))
  INVARIANT
    done = (tick & ((value + 1) mod 2) = 0);
```

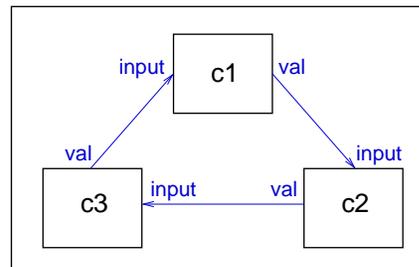
## 3.4 Synchronous vs. Asynchronous

### Synchronous composition

- By default, composition of modules is **synchronous**: *all modules move at each step*.

```
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);
```



### Synchronous composition

A possible execution:

<i>step</i>	<i>c1.val</i>	<i>c2.val</i>	<i>c3.val</i>
0	<b>red</b>	<b>green</b>	<b>blue</b>
1	red	<b>red</b>	<b>green</b>
2	<b>green</b>	red	green
3	green	red	green
4	green	red	<b>red</b>
5	<b>red</b>	<b>green</b>	red
6	red	<b>red</b>	red
7	red	red	red
8	red	red	red
9	red	red	red
10	red	red	red

### Asynchronous composition

- **Asynchronous** composition can be obtained using keyword `process`.
- In asynchronous composition *one process moves at each step*.
- Boolean variable `running` is defined in each process:
  - it is true when that process is selected;
  - it can be used to guarantee a fair scheduling of processes.

```

MODULE cell(input)
VAR  val : {red, green, blue};
ASSIGN next(val) := {val, input};
FAIRNESS running
MODULE main
VAR
  c1 : process cell(c3.val);
  c2 : process cell(c1.val);
  c3 : process cell(c2.val);

```

### Asynchronous composition

A possible execution:

<i>step</i>	<i>running</i>	<i>c1.val</i>	<i>c2.val</i>	<i>c3.val</i>
0	-	<b>red</b>	<b>green</b>	<b>blue</b>
1	c2	red	<b>red</b>	blue
2	c1	<b>blue</b>	red	blue
3	c1	blue	red	blue
4	c2	blue	red	blue
5	c3	blue	red	<b>red</b>
6	c2	blue	<b>blue</b>	red
7	c1	blue	blue	red
8	c1	<b>red</b>	blue	red
9	c3	red	blue	<b>blue</b>
10	c3	red	blue	blue

## 4 Examples

### 4.1 Synchronous

#### 1bit-Adder

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

#### 4bit-Adder

```
MODULE adder(in1, in2)
VAR
  bit[0] : bit-adder(in1[0], in2[0], 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;
```

#### Adder - main

```
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 := in1[0] + 2*in1[1] + 4*in1[2] + 8*in1[3];
  op2 := in2[0] + 2*in2[1] + 4*in2[2] + 8*in2[3];
  sum := a.sum[0] + 2*a.sum[1] + 4*a.sum[2] + 8*a.sum[3]
        + 16*a.overflow;
```

### Adder - simulation

Simulate randomly the system.

- All the variables change their value at every step.
- The initial value of `in1` and `in2` are set randomly and they keep their value throughout the simulation.
- After some (*how many?*) simulation steps, `sum` stores the sum of the two operands.
- After that, no more changes are allowed.

Exercise:

- Add a reset control which changes the values of the operands and restarts the computation of the sum.

### Greatest Common Divisor

Consider the following program:

```
void main() {
    ... // initialization of a and b
    while (a!=b) {
        if (a>b)
            a=a-b;
        else
            b=b-a;
    }
    ... // GCD=a=b
}
```

Remark: Euclid's algorithm for GCD ( $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$ )

### Greatest Common Divisor - labeled

Let's label the *entry point* and the *exit point* of every statement:

```
void main() {
    ... // initialization of a and b
11: while (a!=b) {
    12: if (a>b)
    13: a=a-b;
    else
    14: b=b-a;
    }
15: ... // GCD=a=b
}
```

### Greatest Common Divisor - SMV

Here is the SMV translation:

```
MODULE main()
VAR a: 0..100; b: 0..100;
    pc: {11,12,13,14,15};
ASSIGN
    init(pc):=11;
    next(pc):=
        case
            pc=11 & a!=b: 12;
            pc=11 & a=b: 15;
            pc=12 & a>b: 13;
            pc=12 & a<b: 14;
            pc=13 | pc=14: 11;
            pc=15: 15;
        esac;
```

```
next(a):=
    case
        pc=13: a-b;
        1: a;
    esac;
next(b):=
    case
        pc=14: b-a;
        1: b;
    esac;
```

### Greatest Common Divisor - SMV - correct

Here is the SMV translation:

```
MODULE main()
VAR a: 0..100; b: 0..100;
    pc: {11,12,13,14,15};
ASSIGN
    init(pc):=11;
    next(pc):=
        case
            pc=11 & a!=b: 12;
            pc=11 & a=b: 15;
            pc=12 & a>b: 13;
            pc=12 & a<=b: 14;
            pc=13 | pc=14: 11;
            pc=15: 15;
        esac;
```

```
next(a):=
    case
        pc=13 & a>b: a-b;
        1: a;
    esac;
next(b):=
    case
        pc=14 & b>a: b-a;
        1: b;
    esac;
```

### Greatest Common Divisor - SMV - constraint style

In the constraint style the SMV model looks more like the original:

```
MODULE main
VAR
    a : 0..100; b : 0..100; pc : {11, 12, 13, 14, 15};
INIT pc = 11
TRANS
    pc = 11 -> (((a != b & next(pc) = 12) | (a = b & next(pc) = 15))
                & next(a) = a & next(b) = b)
TRANS
    pc = 12 -> (((a > b & next(pc) = 13) | (a < b & next(pc) = 14))
                & next(a) = a & next(b) = b)
```

```

TRANS
  pc = 13 -> (next(pc) = 11 & next(a) = (a - b) & next(b) = b)
TRANS
  pc = 14 -> (next(pc) = 11 & next(b) = (b - a) & next(a) = a)
TRANS
  pc = 15 -> (next(pc) = 15 & next(a) = a & next(b) = b)

```

## 4.2 Asynchronous

### Simple mutual exclusion

```

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle, entering};
      state = entering & !semaphore : critical;
      state = critical : {critical, exiting};
      state = exiting : idle;
      1 : state;
    esac;

```

### Simple mutual exclusion

*<module user cont'd>*

```

  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
    esac;
FAIRNESS running;

```

```

MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;

```

### Simple mutual exclusion - simulate

Simulate randomly the system:

- At every step, only one process executes.
- The simulation depends on the value of `_process_selector_`.