# NuSMV: Introduction and Examples *

Alessandra Giordani
agiordani@disi.unitn.it
http://disi.unitn.it/~agiordani

Formal Methods Lab Class, April 04, 2014

Università degli Studi di Trento

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le for FM lab 2011/13

# Contents

# Contents

# 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/`

# Contents

# Interactive shell

- `NuSMV -int [filename]` activates an interactive shell
- `read_model [-i filename]` reads the input model.
- `set input_file filename` sets the input model.
- `go` reads and initializes NuSMV for verification or simulation.
- `pick_state [-v] [-r | -i]` picks a state from the set of initial state.
  - `-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` 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.

# Interactive shell

- `reset` resets the whole system (in order to read in another model and to perform verification on it).
- `help` shows the list of all commands (if a command name is given as argument, detailed information for that command will be provided).
- `quit` stops the program.

Argument `-h` prints the command line help.

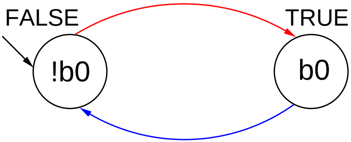
# Inspecting traces

- `goto_state state_label` makes `state_label` the `current state` (it is used to navigate along traces).
- `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.
- `print_current_state [-h] [-v]` prints out the current state.
  - `-v` prints all the variables.

# Contents

# The first SMV program

```
MODULE main
 VAR
   b0 : boolean;

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




An SMV program consists of:

- Declarations of the state variables;
  the state variables determine the state space of the model.
- Assignments that define the valid initial states.
- Assignments that define the transition relation.

# Declaring state variables

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

**boolean**:

```
x : boolean;
```

**enumerative**:

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

**bounded integers** (intervals):

```
n : 1..8;
```

**words**: word types are used to model arrays of bits (booleans) which allow bitwise logical and arithmetic operations.

- `unsigned word[3];`
  vector of 3 bits that allows unsigned operations $[0, 2^3 - 1]$.
- `signed word[7];`
  vector of 7 bits that allows signed operations $[-2^{7-1}, 2^{7-1} - 1]$.

# Arrays

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

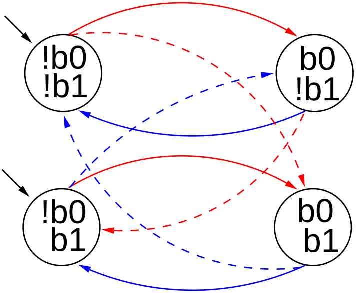
```
VAR
  x : array 0..10 of boolean; -- array of 11 elements
  y : array 2..4 of 0..10;
  z : array 0..10 of array 0..5 of {red, green, orange};
ASSIGN
  init(x[5]) := 1;
  init(y[2]) := {0,2,4,6,8,10}; -- any value in the set
  init(z[3][2]) := {green, orange};
```

Remarks:

- Array indexes in NuSMV *must be constants*;

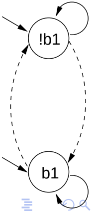
# Adding a state variable

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

ASSIGN
  init(b0) := FALSE;
  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

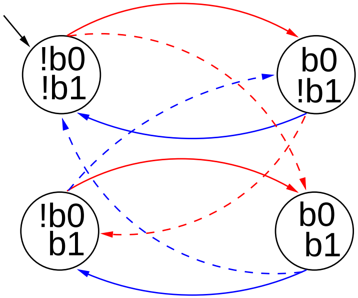
- For each variable, we constrain the values that it can assume in the *initial states*.

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

   init(b1) := FALSE;
```



# Expressions

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

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

  if c1 then e1 else if c2 then e2 else if . . . else en
- Set operators: `{v1,v2,...,vn}` (set expression)
  - `in` (set inclusion) tests a value for membership in a set
  - `union` (set union) takes the union of 2 sets

# Expressions

- Conversion operators:
  - `toint` converts boolean and word to integer.
  - `bool` converts integer and word to boolean (the result of the conversion is `FALSE` if the expression resolves to `0`, `TRUE` otherwise).
  - `swconst` and `uwconst` convert integer to signed and unsigned word respectively.
  - `word1` converts boolean to a single word bit.
  - `unsigned` and `signed` convert signed word to unsigned word and vice-versa.

# 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* (i.e. after each transition).

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

  - `<next_expression>` must evaluate to values in the domain of `<variable>`.
  - `<next_expression>` depends on "current" and "next" variables:

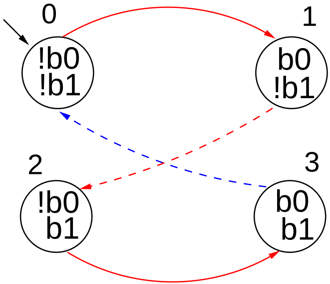
```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

  - 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

A modulo-4 counter:

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

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

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



# Specifying normal assignments

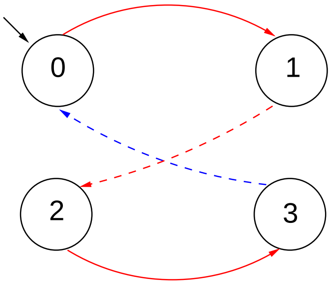
- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

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

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

   out := toint(b0) + 2*toint(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.

- No new state variable is created for defined symbols (hence, no added complexity to model checking).

- Each occurrence of a defined symbol is replaced with the body of the definition.

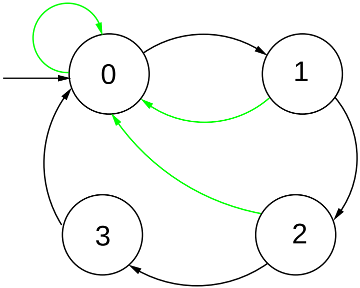
# The DEFINE declaration

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

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

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



# Exercise

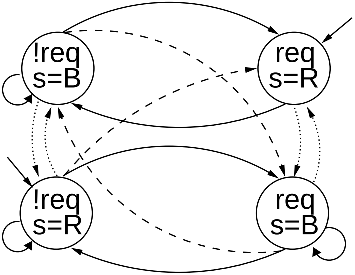
Simulate the system using NuSMV and draw the FSM.

```
MODULE main
VAR
  request : boolean;
  state   : { ready, busy };

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

# Exercise

Simulate the system using NuSMV and draw the FSM.

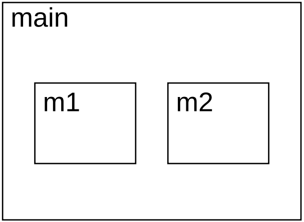
```
MODULE main
VAR
  request : boolean;
  state   : { ready, busy };

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



# 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;
```

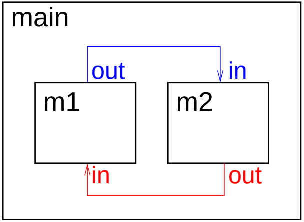



- Modules are instantiated in other modules. The instantiation is performed inside the `VAR` declaration of the parent module.
- In each SMV specification there must be a module `main`.
- All the variables declared in a module instance are referred to via the dot notation (e.g., `m1.out`, `m2.out`).

# 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 : 0..1;
   done  : boolean;

ASSIGN
   init(value) := 0;
   next(value) := case
      tick = FALSE : value;
      tick = TRUE  : (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(TRUE);
   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 `TRUE`.
- 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 : 0..1;
ASSIGN
  init(value) := 0;
  next(value):=
    case
      reset = TRUE : 0;
      TRUE : case
        tick = FALSE : value;
        tick = TRUE  : (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(TRUE, 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;
```

# 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;
      TRUE                    : {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 `ASSIGN`s.

# 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 = TRUE -> FALSE;
      ```
  - *non-determinism is hidden* in the constraints

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

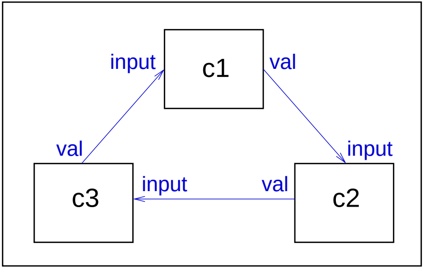
# The modulo 4 counter with reset, using constraints

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

# 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) := input;

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



# Synchronous composition

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 | ... | ... | ... |

# Asynchronous composition

**Asynchronous** composition can be obtained using keyword `process`:
*one process moves at each step*.

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

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.

# Asynchronous composition

**Asynchronous** composition can be obtained using keyword `process`: *one process moves at each step*.

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

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.

In NUSMV 2.5 processes are deprecated!

# Asynchronous composition

A possible execution:

| *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 | c3 | blue | blue | **blue** |

# Contents

# 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 := 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);
```

# 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 (GCD(a, b) = GCD(b, a mod 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
   l1:  while (a!=b) {
     l2:  if (a>b)
       l3: a=a-b;
      else
       l4: b=b-a;
   }
   l5: ... // GCD=a=b
}
```

Here is the SMV translation:

```
MODULE main()
VAR  a: 0..100;  b: 0..100;
  pc: {l1,l2,l3,l4,l5};
ASSIGN
  init(pc):=l1;
  next(pc):=
    case
      pc=l1 & a!=b: l2;
      pc=l1 & a=b: l5;
      pc=l2 & a>b: l3;
      pc=l2 &a<=b: l4;
      pc=l3 | pc=l4: l1;
      pc=l5: l5;
    esac;
```

```
next(a):=
  case
   pc=l3: a-b;
    TRUE: a;
  esac;
next(b):=
  case
   pc=l4: b-a;
    TRUE: 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 : {l1, l2, l3, l4, l5};
INIT pc = l1
TRANS
  pc = l1 -> (((a != b & next(pc) = l2) | (a = b & next(pc) = l5))
              & next(a) = a & next(b) = b)
TRANS
  pc = l2 -> (((a > b & next(pc) = l3) | (a < b & next(pc) = l4))
              & next(a) = a & next(b) = b)
TRANS
  pc = l3 -> (next(pc) = l1 & next(a) = (a - b) & next(b) = b)
TRANS
  pc = l4 -> (next(pc) = l1 & next(b) = (b - a) & next(a) = a)
TRANS
  pc = l5 -> (next(pc) = l5 & next(a) = a & next(b) = b)
```

# 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;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;
FAIRNESS running;
```

# Simple mutual exclusion

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

# Simple mutual exclusion - simulate

Simulate randomly the system:

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