

NuSMV: An Overview

– *Symbolic Model Checking* –

A. Cimatti and M. Pistore

ESSLLI, July 5-9, 2002, Trento (Italy)

Outline

- ➡ The NuSMV project.
- ➡ The SMV language.
- ➡ A demo of NuSMV.

Introduction

- ➡ NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.
- ➡ 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.

History: NuSMV 1

NuSMV is a reimplementaion and extension of SMV.

- ☞ NuSMV started in 1998 as a joint project between ITC-IRST and CMU:
 - the starting point: SMV version 2.4.4.
 - SMV is the first BDD-based symbolic model checker (McMillan, 90).
- ☞ NuSMV version 1 has been released in July 1999.
 - limited to BDD-based model checking
 - extends and upgrades SMV along three dimensions:
 - functionalities (LTL, simulation)
 - architecture
 - implementation
- ☞ Results:
 - used for teaching courses and as basis for several PhD theses
 - interest by industrial companies and academics

History: NuSMV 2

- ☞ The NuSMV 2 project started in September 2000 with the following goals:
 - Introduction of SAT-based model checking
 - OpenSource licensing
 - Larger team (Univ. of Trento, Univ. of Genova, ...)

- ☞ NuSMV 2 has been released in November 2001.
 - first freely available model checker that combines BDD-based and SAT-based techniques
 - extended functionalities wrt NuSMV 1 (cone of influence, improved conjunctive partitioning, multiple FSM management)

- ☞ Results: in the first two months:
 - more than 60 new registrations of NuSMV users
 - more than 300 downloads

OpenSource License

The idea of OpenSource:

- The System is developed by a distributed community
- Notable examples: Netscape, Apache, Linux
- *Potential* benefits: shared development efforts, faster improvements...

Aim: provide a *publicly available, state-of-the-art* symbolic model checker.

- *publicly available*: free usage in research and commercial applications
- *state of the art*: improvements should be made freely available

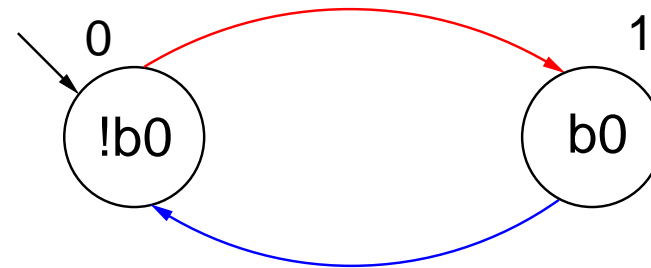
Distribution license for NuSMV 2: *GNU Lesser General Public License (LGPL)*:

- anyone can freely download, copy, use, modify, and redistribute NuSMV 2
- any modification and extension should be made publicly available under the terms of LGPL (“copyleft”)

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 ($\text{init}(b0) := 0$).
- ➡ Assignments that define the transition relation ($\text{next}(b0) := !b0$).

Declaring state variables

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

boolean:

```
VAR
  x : boolean;
```

enumerative:

```
VAR
  st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

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

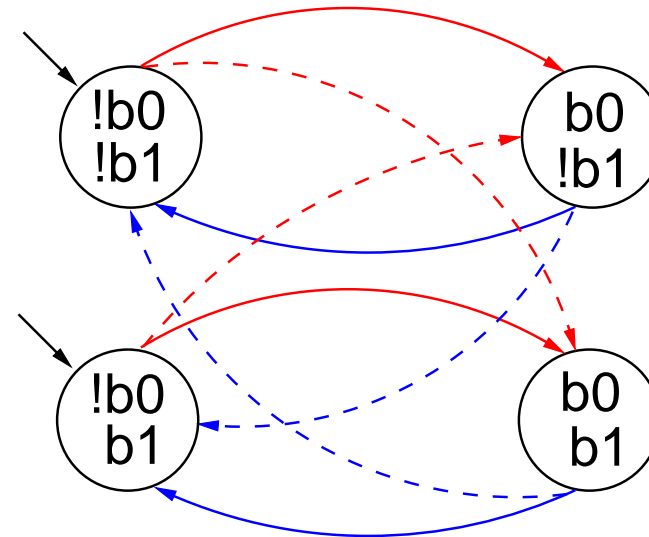
Adding a state variable

```

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

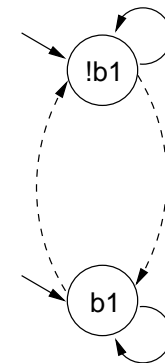
ASSIGN
  init(b0) := 0;
  next(b0) := !b0;

```



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> ;
```

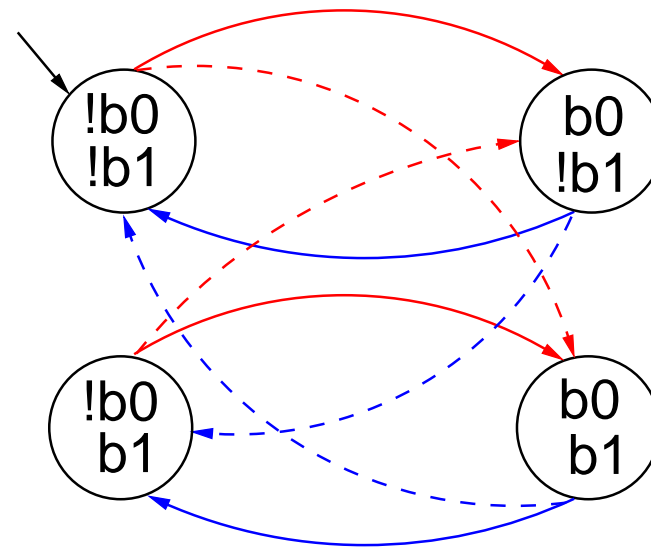
- ➡ <simple_expression> must evaluate to values in the domain of <variable>.
- ➡ If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.

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;
  ...
  l  : 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

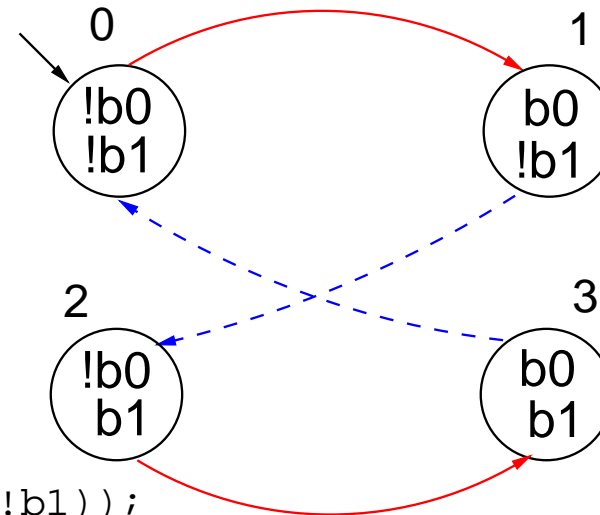
```

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
```

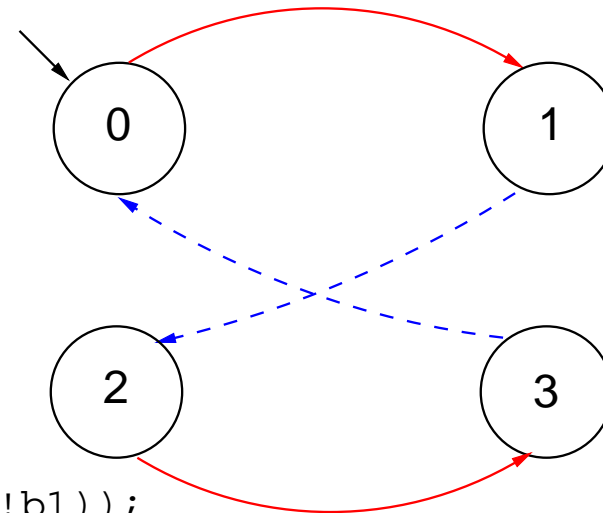
```
  init(b0) := 0;
```

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

```
  init(b1) := 0;
```

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

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



Restrictions on the ASSIGN

For technical reasons, the transition relation must be *total*, i.e., for every state there must be at least one successor state.

In order to guarantee that the transition relation is total, the following restrictions are applied to the SMV programs:

- ➡ 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.

All of the following combinations of assignments are illegal:

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

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

```
status := ready;  
status := busy;
```

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

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

Circular dependencies rule

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

All the following combinations of assignments are illegal:

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

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

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

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

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

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

The following example is *legal*, instead:

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

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

The modulo 4 counter with reset

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

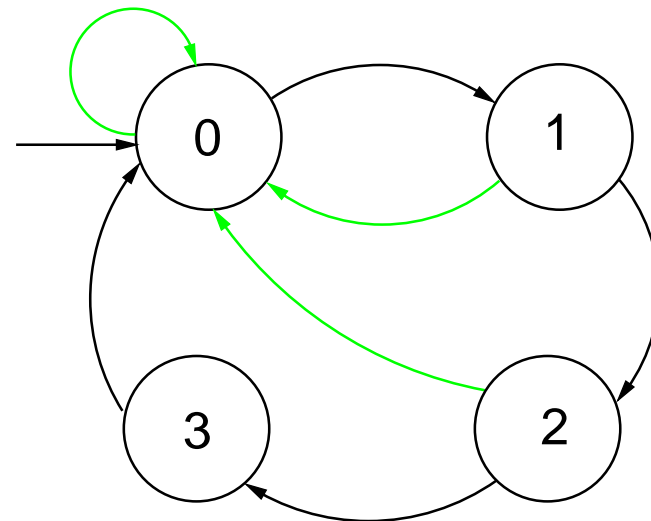
```

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

  ASSIGN
    init(b0) := 0;
    next(b0) := case
      reset = 1 : 0;
      reset = 0 : !b0;
    esac;

    init(b1) := 0;
    next(b1) := case
      reset : 0;
      1     : ((!b0 & b1) | (b0 & !b1));
    esac;

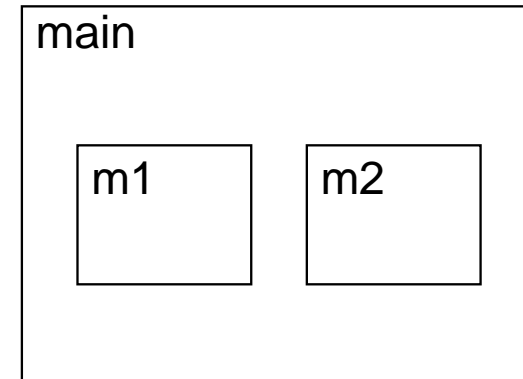
    out := b0 + 2*b1;
  
```



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;
```

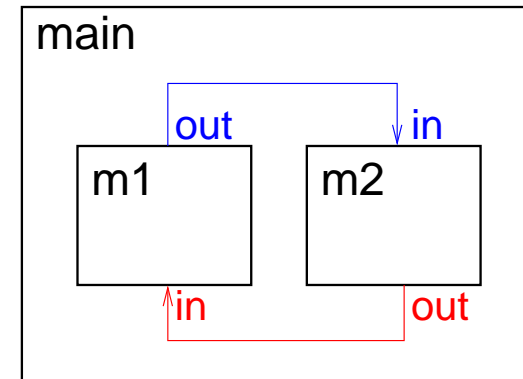


- ➡ Modules are instantiated in other modules. The instantiation is performed inside the `VAR` declaration of the parent module.
- ➡ In each SMV specification there must be a module `main`. It is the top-most module.
- ➡ All the variables declared in a module instance are visible in the module in which it has been instantiated 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 8 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 8 counter revisited

```
MODULE main
  VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;

  ASSIGN
    out := bit0.value + 2*bit1.value + 4*bit2.value;
```

Remarks:

- ➡ Module `counter_cell` is instantiated three 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

A module can contain instances of others modules, that can contain instances of other modules... provided the module references are not circular.

```
MODULE counter_8 (tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out  := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

MODULE counter_512(tick) -- A counter modulo 512
  VAR
    b0 : counter_8(tick);
    b1 : counter_8(b0.done);
    b2 : counter_8(b1.done);
    out : 0..511;
  ASSIGN
    out := b0.out + 8*b1.out + 64*b2.out;
```

Specifications

- ➡ The SMV language allows for the specification of different kinds of properties:
 - invariants,
 - CTL formulas,
 - LTL formulas...
- ➡ Specifications can be added in any module of the program.
- ➡ Each specification is verified separately by NuSMV.

Invariant specifications

- ➡ Invariant properties are specified via the keyword `INVARSPEC`:

```
INVARSPEC <simple_expression>
```

- ➡ Example:

```
MODULE counter_cell(tick)
  ...
MODULE counter_8 (tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

  INVARSPEC
    done <-> (bit0.done & bit1.done & bit2.done)
```

Specifying CTL properties

- ➔ CTL properties are specified via the keyword SPEC:

```
SPEC <ctl_expression>
```

where <ctl_expression> can contain the following temporal operators:

```
AX _ AF _ AG _ A[_ U _]
```

```
EX _ EF _ EG _ E[_ U _]
```

- ➔ It is possible to reach a state in which $out = 3$.

```
SPEC EF out = 3
```

- ➔ A state in which $out = 3$ is always reached.

```
SPEC AF out = 3
```

- ➔ It is always possible to reach a state in which $out = 3$.

```
SPEC AG EF out = 3
```

- ➔ Even time a state with $out = 2$ is reached, a state with $out = 3$ is reached afterwards.

```
SPEC AG (out = 2 -> AF out = 3)
```

Fairness Constraints

Let us consider again the counter with reset.

- ➡ The specification $AF \text{ out} = 1$ is not verified.
- ➡ On the path where `reset` is always 1, then the system loops on a state where `out` = 0, since the counter is always reset:
`reset` = 1,1,1,1,1,1,1...
`out` = 0,0,0,0,0,0,0...
- ➡ Similar considerations hold for the property $AF \text{ out} = 2$. For instance, the sequence:
`reset` = 0,1,0,1,0,1,0...
generates the loop:
`out` = 0,1,0,1,0,1,0...
which is a counterexample to the given formula.

Fairness Constraints

- ➡ NuSMV allows to specify *fairness* constraints.
- ➡ Fairness constraints are formulas which are assumed to be true infinitely often in all the execution paths of interest.
- ➡ During the verification of properties, NuSMV considers path quantifiers to apply only to fair paths.
- ➡ Fairness constraints are specified as follows:

```
FAIRNESS <simple_expression>
```

Fairness Constraints

- ➔ With the fairness constraint

FAIRNESS
out = 1

we restrict our analysis to paths in which the property out = 1 is true infinitely often.

- ➔ The property $AF \text{ out} = 1$ under this fairness constraint is now verified.
- ➔ The property $AF \text{ out} = 2$ is still not verified.
- ➔ Adding the fairness constraint out = 2, then also the property $AF \text{ out} = 2$ is verified.

A demo of NuSMV

In the demo you will see:

- ➡ How to read/modify/save SMV programs.
- ➡ How to execute (simulate) a SMV program.
- ➡ How to verify properties and explore counter-examples of an SMV program.