

# Introduction to NuSMV

Hao Zheng

zheng@cse.usf.edu

Computer Science and Engineering  
University of South Florida

# NuSMV

- NuSMV is a symbolic model checker ( a reimplementaion of the original CMU SMV ).
- The NuSMV input language allows to specify **synchronous** or **asynchronous** systems at **gate** or **behavioral** level.
- It provides constructs for hierarchical descriptions.
  - synchronous modules, or asynchronous processes.
- Systems are modeled as finite state machines.
  - Only finite data types are supported: Boolean, enumeration, array, etc.
- Source: <http://nusmv.irst.itc.it/> for the software and documents.

# Single Process Example

```
-- A simple example.
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state = ready & request = 1 :
        busy;
      1 :
        {ready, busy};
    esac;
```

# Basic Structures of NuSMV Language

- Comments start with "`—`".
- Keywords: **MODULE**, **VAR**, **ASSIGN**, **VAR**, **init**, **next**, **case**, **esac**, etc.
- A descriptions is divided into sections: variable declarations, assignments, etc.
- State space is decided by the number of variables and their types.
- **init** assignments set up the initial state.
  - The initial value of a variable can be any value with the specified type if the variable is not initialized.

# NuSMV Language (con'td)

- **next** assignments describe the state transitions.
- **case** expression includes several branches, and each branch has a condition and a value. It returns the value of a branch if the branch condition is true.
  - Value "{ready, busy}" means that variable "state" can be either "ready" or "busy" in the next state.
  - "request" is non-deterministic since it is not assigned.

# A Binary Counter

```
MODULE counter_cell(carry_in)
  VAR value : boolean;
  ASSIGN
    init(value) := 0;
    next(value) := (value + carry_in) mod 2;
  DEFINE carry_out := value & carry_in;
```

```
MODULE main
  VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
```

# Structural Modeling

- The counter is a connection of three "counter\_cell" instances done as variable declarations.
- Notation 'a.b' is used to access the variables inside a component.
- Keyword **DEFINE** creates an alias for an expression.
  - Can also be done using **ASSIGN**.

# Asynchronous Systems

```
MODULE inverter(input)
  VAR output : boolean;
  ASSIGN
    init(output) := 0;
    next(output) := !input;

MODULE main
  VAR
    gate0 : process inverter(gate3.output);
    gate1 : process inverter(gate1.output);
    gate2 : process inverter(gate2.output);
```

# Asynchronous Systems (cont'd)

- Instances with keyword **process** are composed asynchronously.
  - A process is chosen non-deterministically in a state.
  - Variables in a process not chosen remain unchanged.
- A process may never be chosen.
  - Each process needs fairness constraint "**FAIRNESS** running" to make sure it is chosen infinitely often.

# Running NuSMV: Simulation

- **Simulation** provides some intuition of systems to be checked.
- It allows users to selectively execute certain paths
- Three modes: **deterministic, random, or interactive.**
  - Strategies used to decide how the next state is chosen.
- **Deterministic mode:** the first state of a set is chosen.
- **Random mode:** a state is chosen randomly.
  - Traces are generated in both modes.
  - Traces are the same in different runs with deterministic mode, but may be different with random mode.

# Interactive Simulation

- Users have full control on trace generation.
- Users guide the tool to choose the next state in each step.
  - Especially useful when one wants to inspect a particular path.
- Users are allowed to specify constraints to narrow down the next state selection.
- Refer to section on **Simulation Commands** in the User Manual.

# Model Checking

- Decides the truth of CTL/LTL formulas on a model.
- **SPEC** is used for CTL formulas, while **LTLSPEC** is used for LTL formulas.
- A counter-example (CE) may be generated if a formula is false.
  - CE cannot be generated for formula with E quantifier.

# NuSMV CTL Specification

- Introduced with **SPEC** for each module.
- Plain CTL extended with real-time.
  - Each state transition takes unit amount of time.
- $[A|E]BG^{m..n} f$  :  $f$  holds from the  $m$ th state until the  $n$ th state from the current state on *all* or *some* paths.
- $[A|E]BF^{m..n} f$  :  $f$  holds in any state within from the  $m$ th state and the  $n$ th state from the current state on *all* or *some* paths.
- $[A|E](f_1BU^{m..n} f_2$  :  $f_2$  holds in state  $s_i$  such that  $m \leq i \leq n$ , and  $f_1$  holds in all state  $s_j$  such that  $m \leq j < i$  from the current state on *all* or *some* paths.

# NuSMV LTL Specification

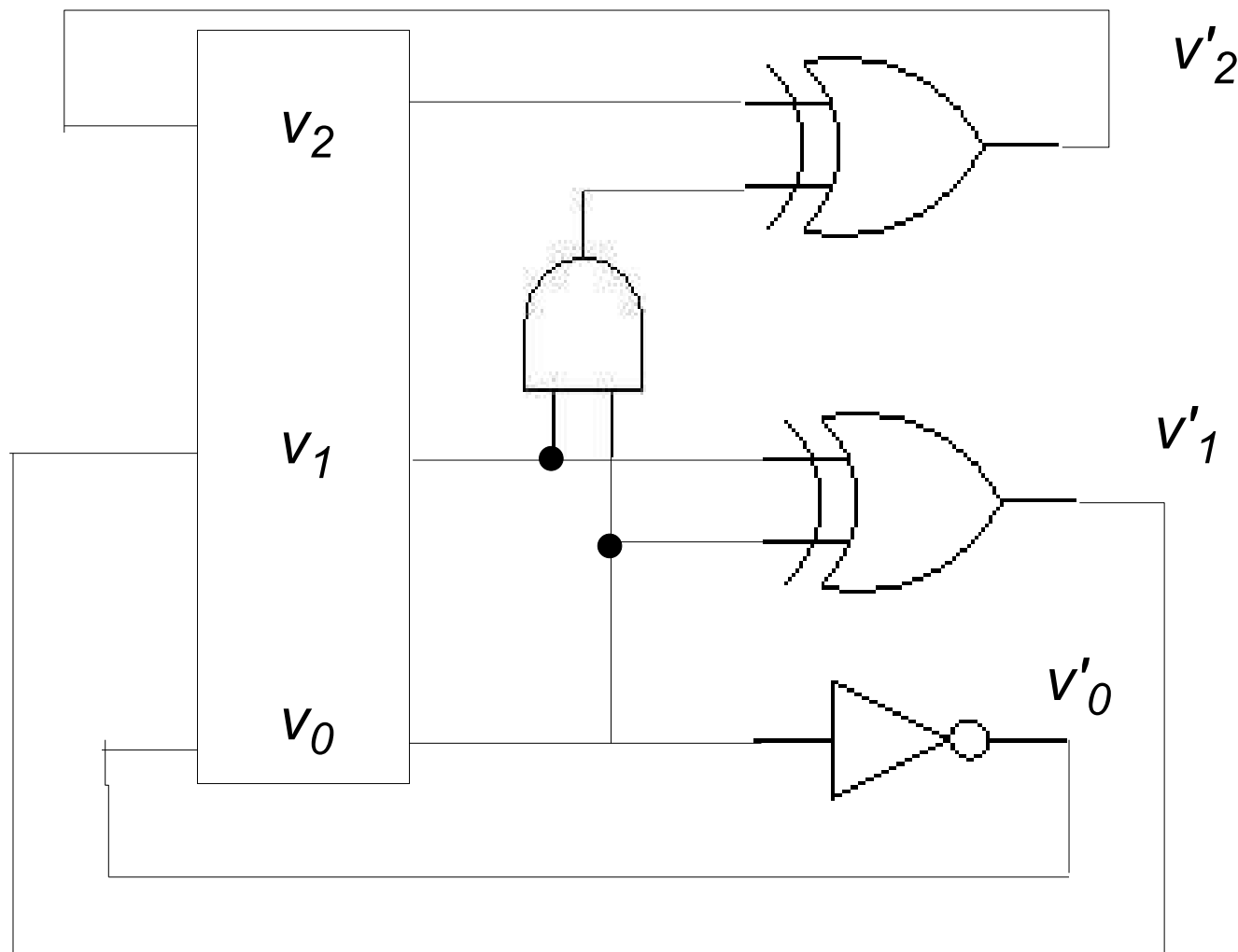
- Introduced with **LTLSPEC** for each module.
- Used for complete or bounded model checking.
- Includes **past** temporal operators in addition to the other usual temporal operators.
- $\Upsilon f$  holds if  $f$  holds in the immediate previous state.
- $Hf$  holds if  $f$  holds in all previous states.
- $Of$  holds if  $f$  holds in a previous state.
- $fSg$  holds if  $f$  holds in all states until now following the state where  $g$  holds.

# A 3-bit Counter: Functional Modeling

- When *reset* is asserted, *output* goes to 0.
- Otherwise, *output* increments by 1 in each cycle.

```
MODULE counter(reset)
VAR
    output : 0..7;
ASSIGN
    init(output) := 0;
    next(output) :=
        case
            reset = 1 : 0;
            output < 7 : output + 1;
            output = 7 : 0;
            1 : output;
        esac;
```

# A 3-bit Counter: Gate-level Modeling



# A 3-bit Counter: Gate-level Modeling

```
MODULE counter(reset)
VAR
  v0 : boolean;  v1 : boolean;  v2 : boolean;
ASSIGN
  next(v0) := case
    reset = 1 : 0;
    1 : !v0;
  esac;
  next(v1) := case
    reset = 1 : 0;
    1 : v0 xor v1;
  esac;
  next(v2) := case
    reset = 1 : 0;
    1 : (v0 & v1) xor v2;
  esac;
```

# A 3-bit Counter: Model Checking

```
MODULE main
VAR
    reset : boolean;
    dut    : counter(reset);
ASSIGN
    init(reset) := 1;
DEFINE
    cnt_out = dut.output;

SPEC AG(reset -> cnt_out=0 )
SPEC AG(!reset & cnt_out=0 -> AX(cnt_out=1))
...
SPEC AG(!reset & cnt_out=7 -> AX(cnt_out=0))
```

# A SR-Latch: Functional Modeling

- It has two inputs  $S$  and  $R$ , and two outputs  $Q$  and  $NQ$ .
- When  $S = 1$  and  $R = 0$ ,  $Q = 1$  and  $NQ = 0$ .
- When  $S = 0$  and  $R = 1$ ,  $Q = 0$  and  $NQ = 1$ .
- When  $S = 0$  and  $R = 0$ ,  $Q$  and  $NQ$  remain unchanged.
- Otherwise,  $S = 1$  and  $R = 1$  should be avoided.

# A SR-Latch: Functional Modeling (1)

```
MODULE SR(S, R)
VAR
    Q : boolean;
    NQ : boolean;
ASSIGN
    init(Q) := 0;
    next(Q) :=
        case
            R = 1 : 0;
            S = 1 : 1;
            1 : Q;
        esac;
    NQ := !Q;
```

# A SR-Latch: Functional Modeling (2)

```
MODULE SR_Q(S, R)
VAR
    Q : boolean;
ASSIGN
    init(Q) := 0;
    next(Q) :=
        case
            R = 1 : 0;
            S = 1 : 1;
            1 : Q;
        esac;
```

# A SR-Latch: Functional Modeling (2)

```
MODULE SR_NQ(S, R)
VAR
    NQ : boolean;
ASSIGN
    init(NQ) := 0;
    next(NQ) :=
        case
            R = 1 : 1;
            S = 1 : 0;
            1 : NQ;
        esac;
```

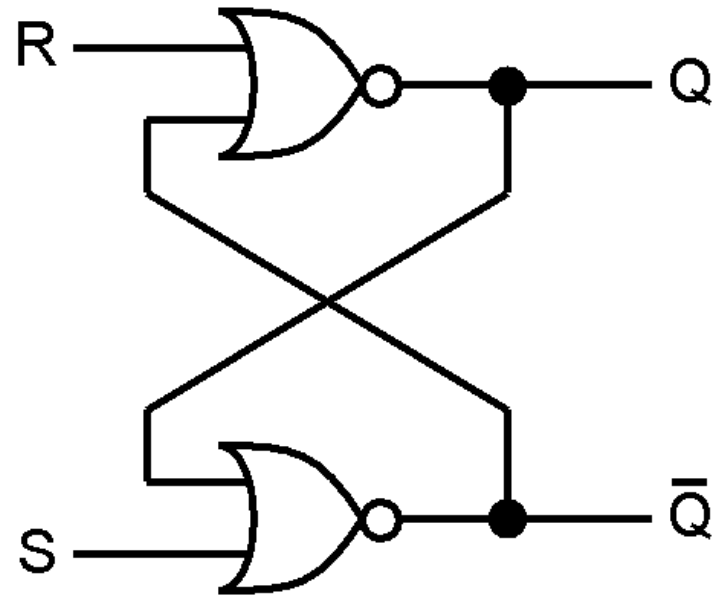
# A SR-Latch: Functional Modeling (3)

```
MODULE SR(S, R)
VAR
    q : process SR_Q(S, R);
    nq : process SR_NQ(S, R);

-- correctness requirement
SPEC AG( q.Q = !nq.NQ )

-- environment assumption
INVAR !(S & R) )
```

# A SR-Latch: Gate-Level Modeling



# A SR-Latch: Gate-Level Modeling

```
MODULE NOR2(a, b)
  VAR
    output : boolean;
  ASSIGN
    init(output) := 0;
    next(output) :=
      case
        a | b : 0;
        1 : 1;
      esac;
```

# A SR-Latch: Gate-Level Modeling

```
MODULE SRL(S, R)
  VAR
    Q : boolean;
    NQ : boolean;
    nor1 : process NOR2(R, NQ);
    nor2 : process NOR2(S, Q);
  ASSIGN
    Q := nor1.output;
    NQ := nor2.output;

  SPEC AG( S -> AX (Q & !NQ) )
  SPEC AG( R -> AX (!Q & NQ) )
```