Model Checker NuSMV

Hao Zheng

Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
Email: zheng@cse.usf.edu
Phone: (813)974-4757
Fax: (813)974-5456

Overview

NuSMV

- NuSMV is a symbolic model checker (a reimplementation 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 date types are supported: Boolean, enumeration, array, etc.
- Source: http://nusmv.irst.itc.it/ for the software and documents.
  - User manuals, tutorials, etc.

Contents

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- Comments start with “- -”.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- Top level model is “main”.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- Each module is divided into section starting with **VAR**, **ASSIGN**, **SPEC**, etc

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- State space of a module is defined by the variables and their types.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- Section **ASSIGN** defines the initialization and transition relations of variables.
  - **init**(v) initializes a variable. An uninitialized variable can take any value of its type.
  - **next**(v) defines the next state of v based on current states.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- **case** expression includes several branches, each of which returns a value if the branch condition is true.
- If multiple brach conditions are true, one is selected non-deterministically.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- If a variable is not assigned by **next**(v), its next state is selected non-deterministically from its type.
- See `request`.

Single Process 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;
SPEC AG (request -> AF (state = busy))
```

- Section **SPEC** includes CTL formulas.
- Section **LTLSPEC** includes LTL formulas.

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);
```

- The counter is a connection of three `counter_cell` instances done as variable declarations.
- A module instance can take parameters.
- Notation `a.b` is used to access the variables inside a component.

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);
```

- 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;
  FAIRNESS running

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

- Instances with keyword **process** are composed asynchronously.
  - A process is chosen non-deterministically in a state.
  - Variables in a process not chosen remain unchanged.

Asynchronous Systems

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

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

- A process may never be chosen.
  - Each process needs fairness constraint "**FAIRNESS** running" to make sure it is chosen infinitely often.

Asynchronous Systems (cont'd)

- Keyword **process** may be going away.

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

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

- Use keyword **union** to allow each variable to nondeterministically change or keep the current value.
- Cannot enforce fairness.

Direct Specification

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

MODULE inverter(input) VAR
   output : boolean;
INIT
   output = FALSE;
TRANS
   next(output) = !input | next(output) = output;
```

- The set of initial states is specified as a formula in the current state variables (**INIT**)

Direct Specification

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

MODULE inverter(input) VAR
   output : boolean;
INIT
   output = FALSE;
TRANS
   next(output) = !input | next(output) = output;
```

- The transition relation is specified as a propositional formula in terms of the current and next state variables (**TRANS**).
- In the example, each gate can choose non-deterministically

Contents

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.

Contents

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.
  - CTL operators: AG, AF, AX, A(f U g), EG, EF, EX, E(f U g),
- Plain CTL extended with real-time.
  - Each state transition takes unit amount of time.
- [A | E]BG$m..n\ f$: f holds from the mth state until the nth state from the current state on *all* or *some* paths.
- [A | E]BF$m..n\ f$: f holds in any state within from the mth state and the nth state from the current state on *all* or *some* paths.
- [A | E]$(f_1$BU$m..n\ f_2$: $f_2)$ 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.
- Refer to the NuSMV User Manual for detailed description.

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.
- $\mathsf{Y}f$ holds if f holds in the immediate previous state.
- $\mathsf{H}f$ holds if f holds in all previous states.
- $\mathsf{O}f$ holds if f holds in a previous state.
- $f\mathsf{S}g$ holds if f holds in all states until now following the state where g holds.
- Refer to the NuSMV User Manual for detailed description.

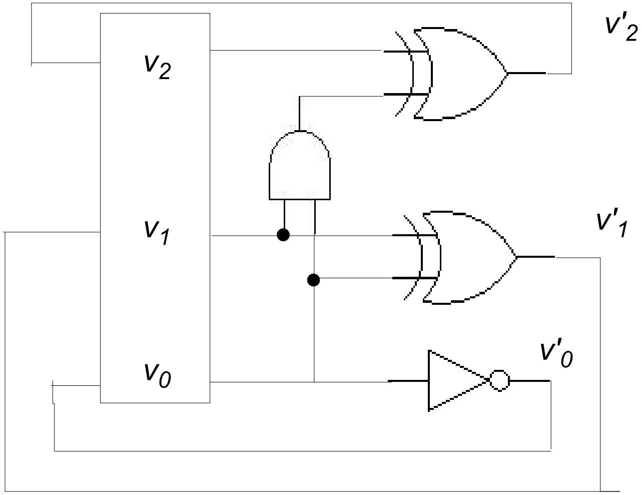
Contents

A 3-bit Counter: Functional Modeling

- When $reset$ is asserted, $output$ goes to 0.
- Ohterwise, $output$ 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;
```

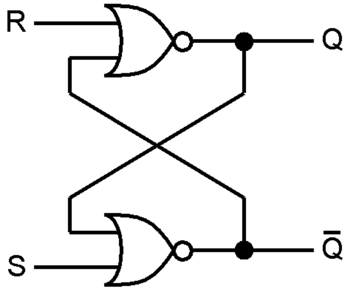
A SR-Latch: Functional Modeling (3)

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

-- correctnes 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) )
```

Semaphore

```
MODULE main
VAR
   semaphore : boolean;
   proc1 : process user(semaphore);
   proc2 : process user(semaphore);
ASSIGN
   init(semaphore) := FALSE;
SPEC AG !(proc1.state = critical & proc2.state = critical)
SPEC AG (proc1.state = entering -> AF proc1.state = critical)
```

Semaphore

```
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
```

Semaphore – LTL Model Checking

```
MODULE main
VAR
   semaphore : boolean;
   proc1 : process user(semaphore);
   proc2 : process user(semaphore);
ASSIGN
   init(semaphore) := FALSE;
LTLSPEC G !(proc1.state = critical & proc2.state = critical)
LTLSPEC G (proc1.state = entering -> F proc1.state = critical)
```

From Promela to NuSMV

```
#define true 1             /* spinroot: file ex.4b */
#define false 0
bool flag[2];
bool turn;

active [2] proctype user()
{ flag[_pid] = true;
     turn = _pid;
    (flag[1-_pid] == false || turn == 1-_pid);
crit: skip; /* critical section */
    flag[_pid] = false
}
```

From Promela to NuSMV

```
MODULE main
VAR
   flag : array 0..1 of boolean;
   turn : boolean;
   proc1 : user(flag, turn, 0);
   proc2 : user(flag, turn, 1);
ASSIGN
   init(flag[0]) := FALSE;
   init(flag[1]) := FALSE;
   init(turn) := FALSE;
```

From Promela to NuSMV

```
MODULE user(flag, turn, pid)
VAR
   state : {s6, s7, s8, s9, s10};
ASSIGN
   init(state) := s6;
   next(state) :=
      case
         state = s6 : {s6, s7};
         state = s7 : {s7, s8};
         state = s8 & (flag[1-pid] = FALSE | turn = 1-pid) : {s8, s9};
         state = s9 : {s9, s10};
         state = s10 : {s10, s6};
         TRUE : state;
      esac;
```

From Promela to NuSMV

```
next(flag[pid]) :=
   case
      state = s6    : TRUE;
      state = s10   : FALSE;
      TRUE          : flag[pid];
   esac;

next(turn) :=
   case
      state = s7    : pid;
      TRUE          : turn;
   esac;
```