Model-based Test Generation using NuSMV

Dongwoo Kim
Kyungpook National University
kdw9242@gmail.com

KNU KYUNGPOOK NATIONAL UNIVERSITY

Software Safety Engineering LAB

NuSMV

- NuSMV is a symbolic model checker
  OpenSource tool
  Free Software license.

- NuSMV home page: http://nusmv.fbk.eu/

✓ Modelling the system

✓ Modelling the properties

✓ Verification

  ✓ simulation

  ✓ checking of formulae

NuSMV 2.6 documents & references

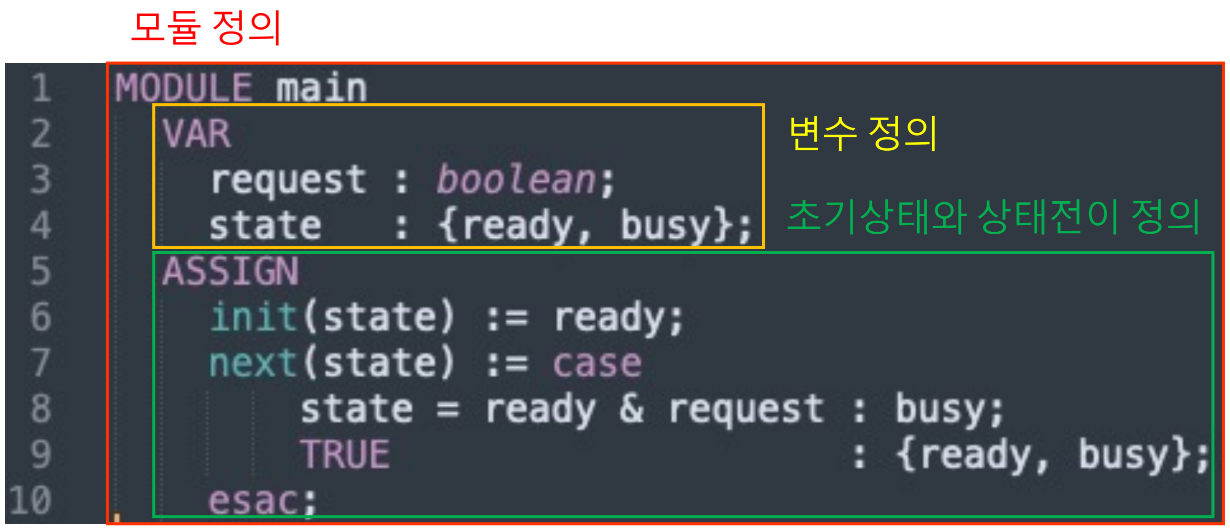
- NuSMV 2.6 Tutorial.
  R. Cavada, A. Cimatti et al., FBK-IRST
  Distributed archive of NuSMV (/share/nusmv/doc/tutorial.pdf)
- NuSMV 2.6 User Manual.
  R. Cavada, A. Cimatti et al., FBK-IRST
  Distributed archive of NuSMV (examples/nusmv.pdf)
- Examples are available in the archive of NuSMV.
- Examples are available also at the URL <http://nusmv.fbk.eu/examples/examples.html>
- ppt slides: Cinzia Bernardeschi, Model checking, 2019
- Contents & examples below are taken from the documents.

NuSMV Modelling language

- A system is a program that consists of one or more modules.
- A module consists of
  - a set of state variables;
  - a set of initial states;
  - a transition relation defined over states.
- Every program starts with a module named MAIN
- modules are instantiated as variables in other modules
- Modules can be Synchronous or Asynchronous

A simple program

- A system can be ready or busy. Variable state is initially set to ready. Variable request is an external uncontrollable signal. When request is TRUE and variable state is ready, variable state becomes busy. In any other case, the next value of variable state can be ready or busy: request is an unconstrained input to the system.



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

Data types

- The language provides the following types
    - Booleans
    - enumerations - ex) {OK, y, z}
    - bounded integers
    - words: unsigned word[.] and signed word[.] types are used to model vector of bits (Booleans) which allow bitwise logical and arithmetic operations (unsigned and signed)
    - Arrays
      lower and upper bound for the index, and the type of the elements
      ex) array 0..3 of Boolean
      ex) array 10..20 of {OK, y, z}
    - .....

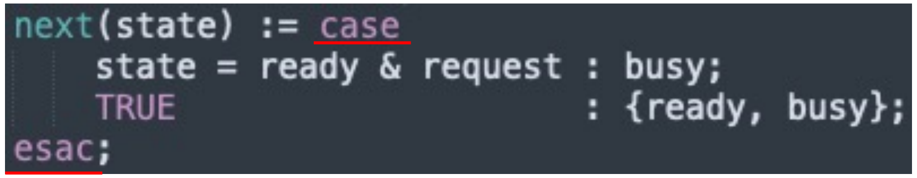
Operators

- Logical and Bitwise &, |, xor, xnor, ->, <->
- Equality (=) and Inequality (!=)
- Relational Operators >, <, >=, <=
- Arithmetic Operators +, -, * , /
- mod (algebraic remainder of the division)
- Shift Operators «, »
- Index Subscript Operator [ ]
- .....

Other expressions

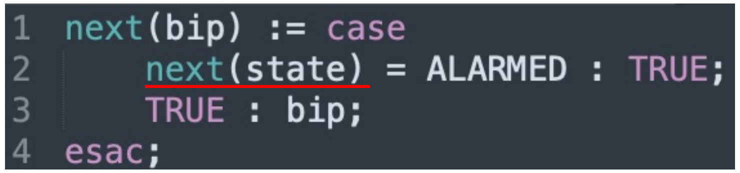
- Case expression

```
case
  cond1 : expr1;
  cond2 : expr2;
  cond3 : expr3;
  ...
  TRUE  : exprN;
esac;
```



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

- Next expression
  refer to the values of variables in the next state
  next(v) refers to that variable v in the next time step
  next((1 + a) + b) is equivalent to **(1 + next(a)) + next(b)**
  <u>**next** operator cannot be applied twice, i.e. next(next(a))</u>



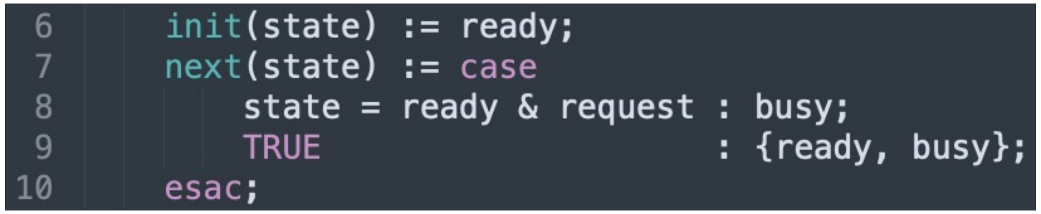
```
next(bip) := case
     next(state) = ALARMED : TRUE;
     TRUE : bip;
esac;
```

Finite state machine – FSM

- Transition relation describing how inputs leads from one state to possibly many different states

- Initial state: `init(<variable>) := <simple_expression> ;`
  variables not initialized can assume any value in the domain of the type of the variable

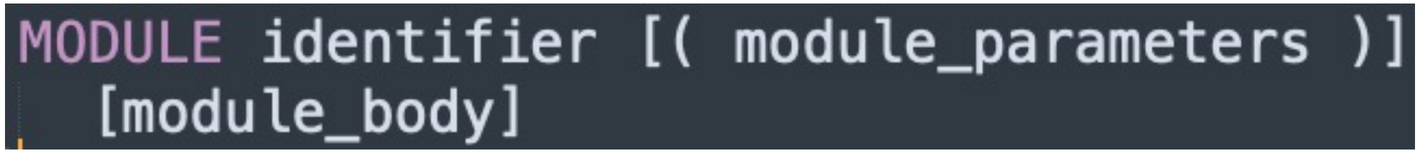
- Transition relation: `next(<variable>) := <simple_expression> ;`
  simple_expression gives the value of the variable in the next state of the transition system



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

Module declaration

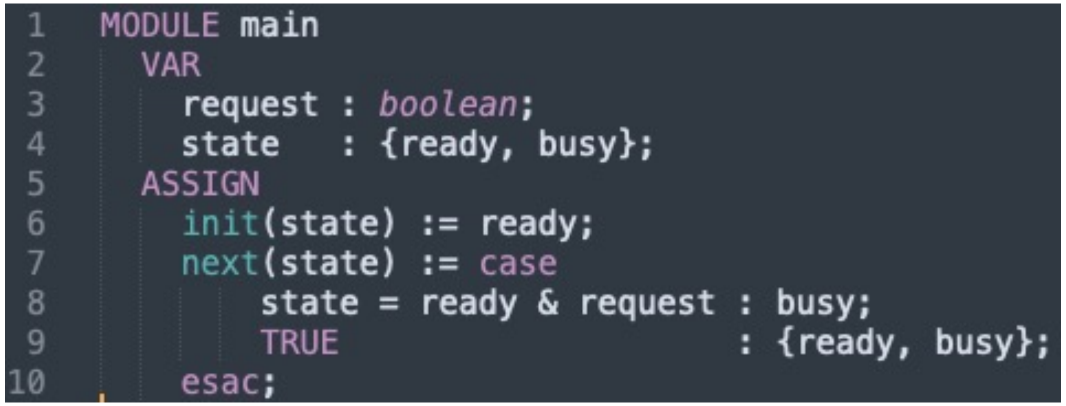
- A module declaration is a collection of declarations, constraints and specifications (logic formulae).
- A module can be reused as many times as necessary. Modules are used in such a way that each instance of a module refers to different data structures.
- A module can contain instances of other modules, allowing a structural hierarchy to be built.



```
MODULE identifier [( module_parameters )]
  [module_body]
```

A simple program

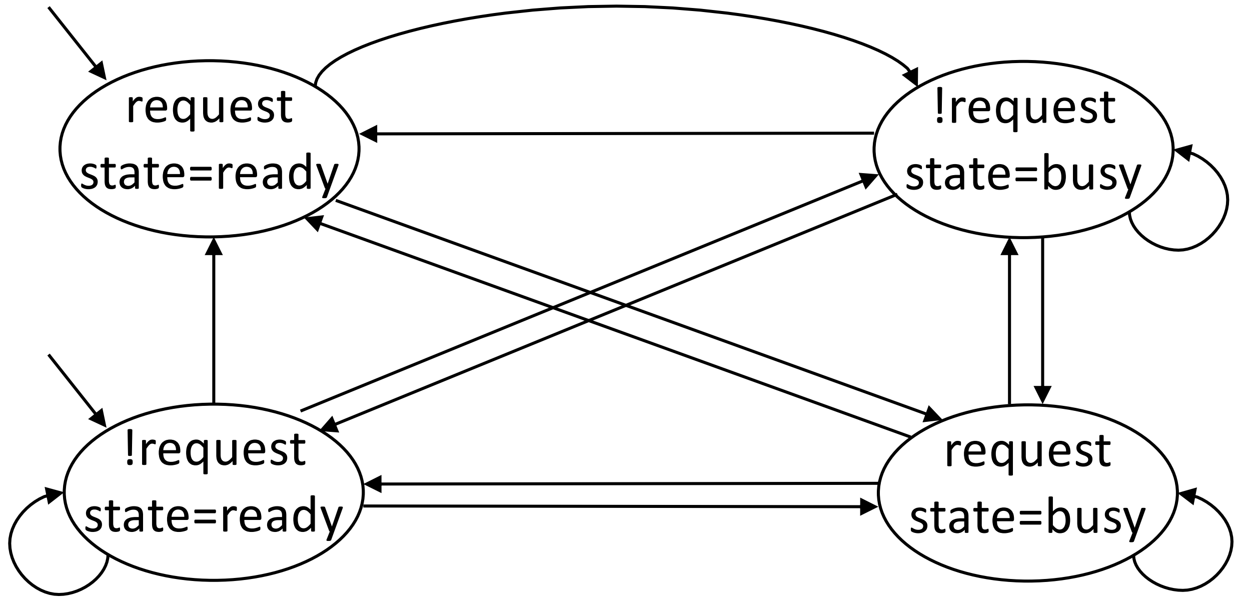
- A system can be ready or busy. Variable state is initially set to ready. Variable request is an external uncontrollable signal. When request is TRUE and variable state is ready, variable state becomes busy. In any other case, the next value of variable state can be ready or busy: request is an unconstrained input to the system.



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

A simple program - continue

- Build the transition system (also named Finite state machine - FSM)
  4 states
  2 initial states
  14 transitions



NuSMV Installation

NuSMV usage

Running NuSMV

```
./NuSMV -int
  activates an interactive shell for simulation

read_model [-i filename]
  reads the input model

go
  reads and initializes NuSMV for simulation

reset
  resets the whole system

help
  shows the list of all commands

quit
  stops the program
```

Simulation

```
pick_state [-v] [-r | -i]
  picks a state from the set of initial states -v prints the chosen state.
  -r pick randomly
  -i pick interactively
  -c pick state having specific constraints

simulate [-p | -v] [-r | -i] -k
  generates a sequence of at most k steps starting from the current state
  -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

show_traces [-v] [trace number]
  shows the trace identified by trace number or the most recently generated trace.
  -v prints prints all the state variables.

print_current_state [-v]
  prints out the current state. -v prints all the variables
```

Simulation – usage example

An interactive session

```
./NuSMV -int

read_model -i simple.smv
go
pick_state -r
print_current_state -v
simulate -v -r -k 3
show_traces -t
show_traces -v

—
pick with constraint
pick_state -c "request = TRUE" -i
```

Verification / Test generation

- Specifications written in CTL can be checked on the FSM .
- OPERATORS:

  -
    ```
    EX p
    AX p
    EF p
    AF p
    EG p
    AG p
    E[p U q]
    A[p U q]
    ```
- A CTL formula is true if it is true in all initial states.

Checking properties

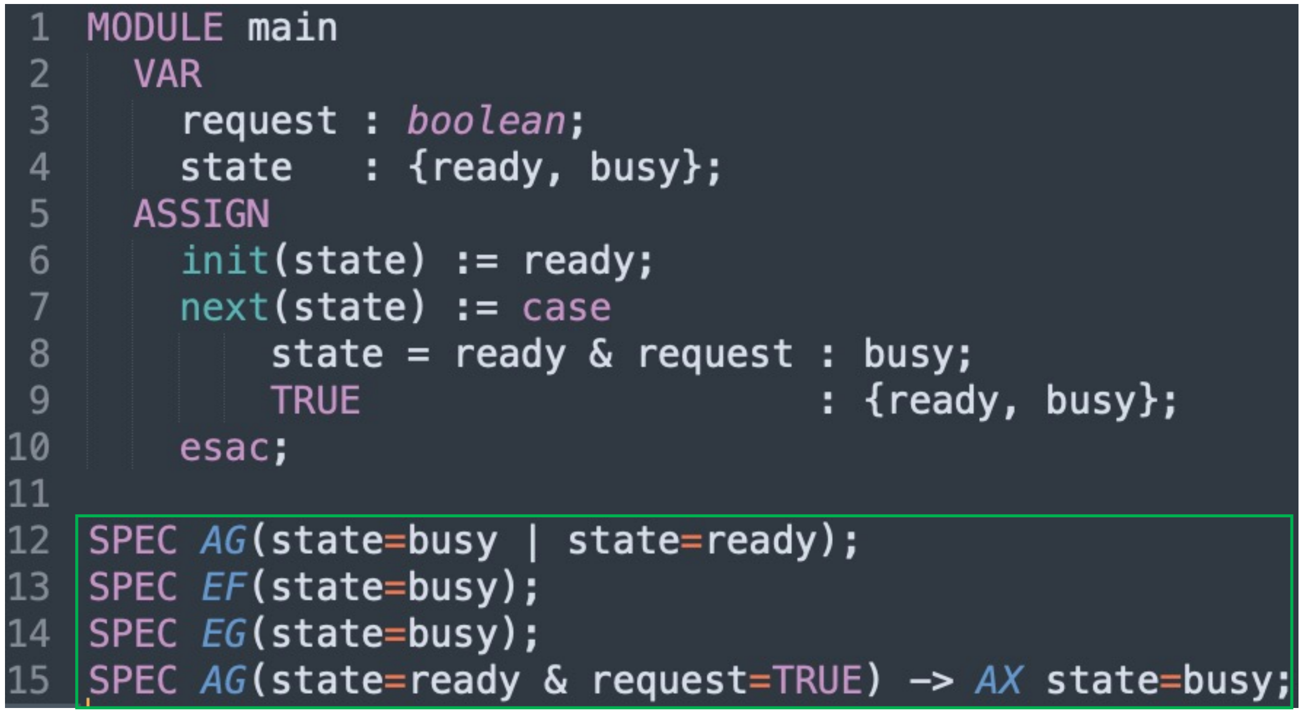
1. Specify the formula:

```
-- model
MODULE ...
...

-- property
SPEC <CTL_formula>
```

2. Invoke NuSMV as follows:

```
./NuSMV file.smv
```



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

SPEC AG(state=busy | state=ready);
SPEC EF(state=busy);
SPEC EG(state=busy);
SPEC AG(state=ready & request=TRUE) -> AX state=busy;
```

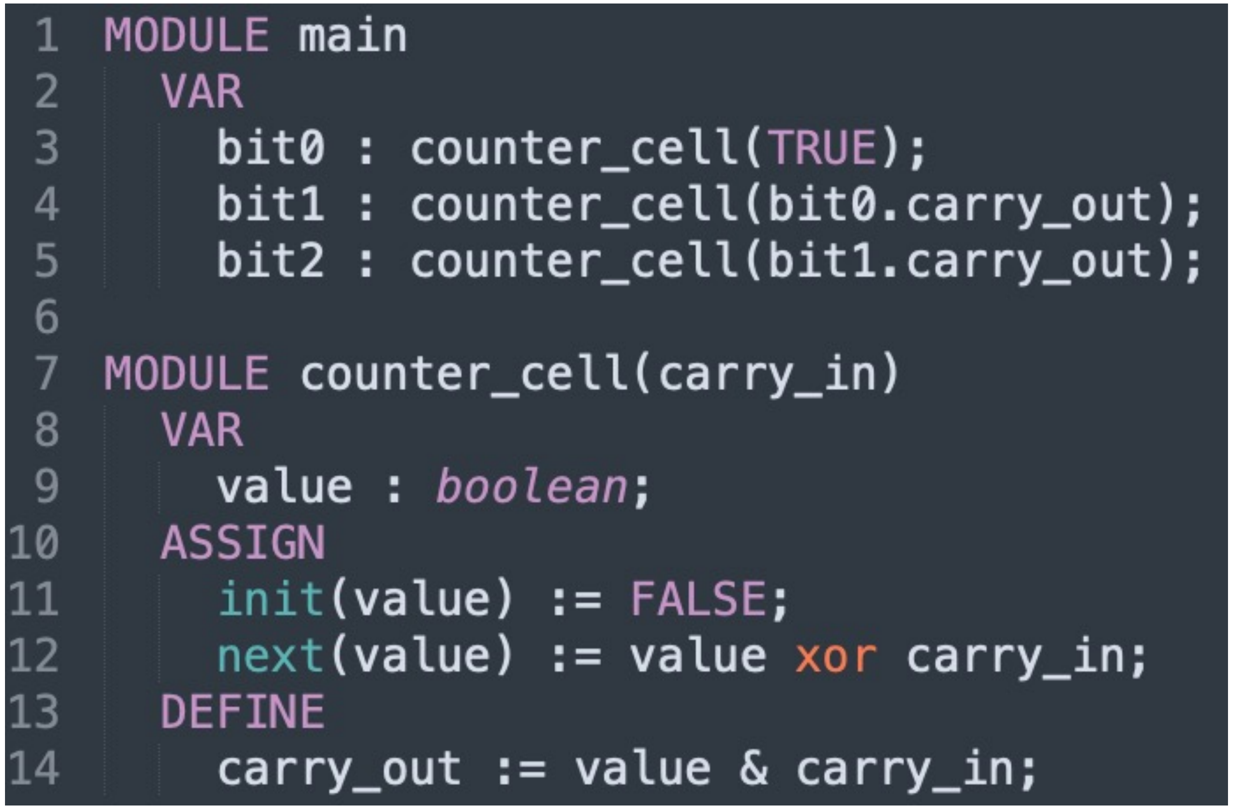
A system with more than one module

- An instance of a module is created using the VAR declaration.
- In the declaration, actual parameters are specified
- Module declarations may be parametric.
- variables declared in a module are local to the module
- synchronous composition: all modules move at each step (by default)

```
MODULE main
  VAR
    a : boolean;
    b : foo(a);

MODULE foo(x)
  VAR
    value : boolean;
  ASSIGN
    next(value) := x;
```

Another example



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

MODULE counter_cell(carry_in)
  VAR
    value : boolean;
  ASSIGN
    init(value) := FALSE;
    next(value) := value xor carry_in;
  DEFINE
    carry_out := value & carry_in;
```

Find a property for overflow scenario

Model-based Test Generation for OSEK/VDX Operating Systems Demo