NuSMV 2.5 tutorial

with a gentle introduction to model checking

Formal Methods in Software Development
Master Degree, 2017/2018
Prof. Anna Labella

Dr. Federico Mari
`mari.di.uniroma1.it` – `mari@di.uniroma1.it`

Sapienza
Università di Roma

Model Checking Laboratory Group
`http://mclab.di.uniroma1.it/`

Computer Science Department
Sapienza University of Rome

April 16, 2018

Introduction

- Model Checking
- NuSMV overview

Input language by examples

- Synchronous systems
- Asynchronous systems
- Direct specification

Simulation

- Trace strategies
- Interactive mode

CTL model checking

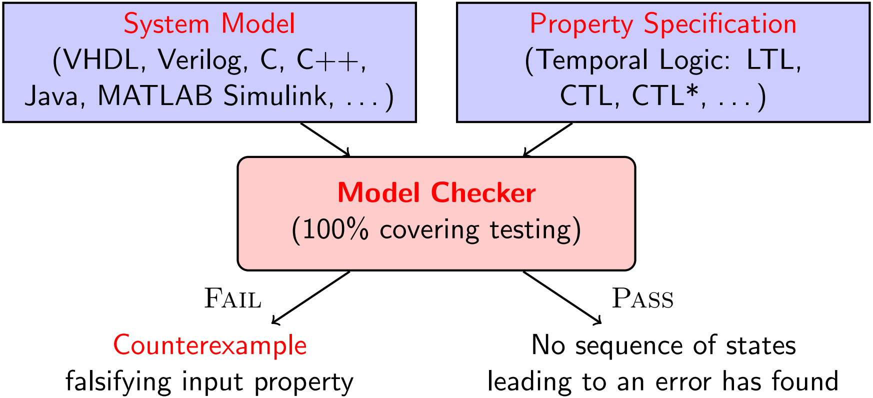
LTL model checking

- Semaphore example
- Past temporal operators

Bounded model checking

- Introduction
  - Model Checking
  - NuSMV overview
- Input language by examples
  - Synchronous systems
  - Asynchronous systems
  - Direct specification
- Simulation
  - Trace strategies
  - Interactive mode
- CTL model checking
- LTL model checking
  - Semaphore example
  - Past temporal operators
- Bounded model checking

Model Checking



System Model
(VHDL, Verilog, C, C++,
Java, MATLAB Simulink, . . .)

Property Specification
(Temporal Logic: LTL,
CTL, CTL*, . . .)

Model Checker
(100% covering testing)

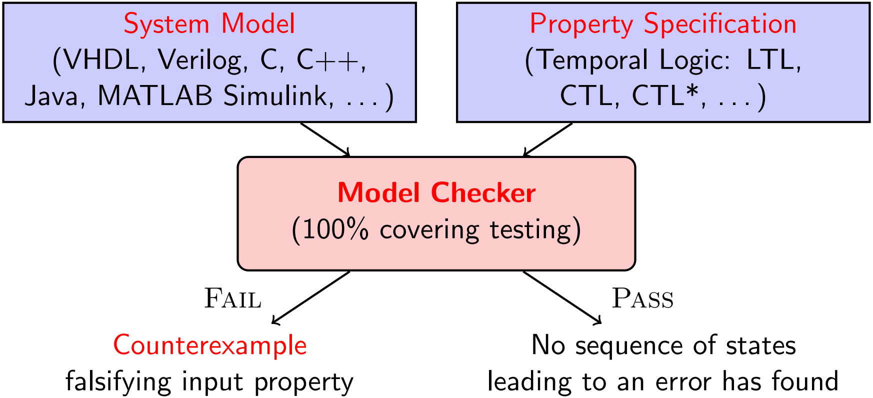
FAIL

PASS

Counterexample
falsifying input property

No sequence of states
leading to an error has found

Model Checking



System Model
(VHDL, Verilog, C, C++,
Java, MATLAB Simulink, . . .)

Property Specification
(Temporal Logic: LTL,
CTL, CTL*, . . .)

Model Checker
(100% covering testing)

FAIL

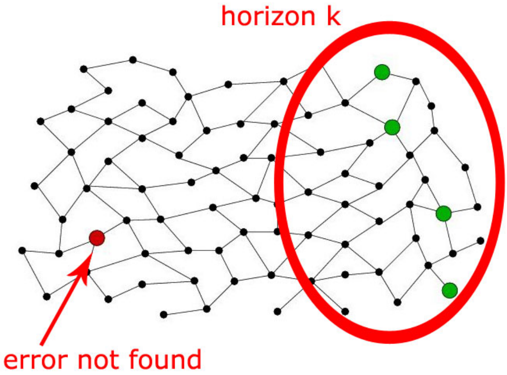
Counterexample
falsifying input property

PASS

No sequence of states
leading to an error has found

Problem: State Explosion! → Abstractions, Bounded model checking, . . .

Bounded Model Checking



Runs of length at most k

Model checkers

Model checkers can be

- Explicit Perform explicit state space enumeration and property checking in each state
- Symbolic Instead of explicitly enumerating all possible states, the transition relation is represented as formulas, binary decision diagrams (BDD) or other related data structures

Depending on the domain a type could be more efficient than the other (e.g. explicit for protocols, symbolic for circuits, ...)

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Overview

NuSMV[a]

- is a symbolic model checker developed by FBK-IRST, CMU, Univ. Trento
- is a reimplementation and extension of SMV, the first model checker based on BDDs
- combines BDD-based model checking (CUDD library) and SAT-based model checking (Minisat and/or ZChaff SAT Solvers)
- is the base of NashMV[b]

Official logo[a]

[a] http://nusmv.fbk.eu/

[b] http://mclab.di.uniroma1.it/site/index.php/software/19-nashmv

- Introduction
  - Model Checking
  - NuSMV overview
- Input language by examples
  - Synchronous systems
  - Asynchronous systems
  - Direct specification
- Simulation
  - Trace strategies
  - Interactive mode
- CTL model checking
- LTL model checking
  - Semaphore example
  - Past temporal operators
- Bounded model checking

Input language by examples

- A complete description of the NuSMV language can be found in the NuSMV 2.5 User Manual[1]
- All mentioned example files can be found in the distributed archive of NuSMV 2.5[2]
- Description of Finite State Machines (FSMs)–synchronous or completely asynchronous, detailed or abstract–via the definition of the transition relation (valid evolutions of the FSM)
- Modular hierarchical descriptions, definition of reusable components
- Available types for variables
  - finite ones (booleans, scalars and fixed arrays)
  - static data types can be defined

[1] http://nusmv.fbk.eu/NuSMV/userman/index-v2.html

[2] http://nusmv.fbk.eu/examples/examples.html

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Single process example I

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

Single process example II

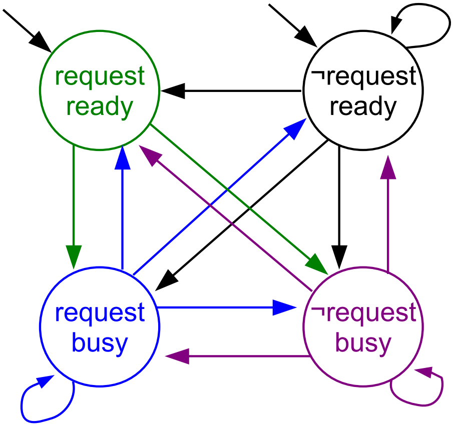
```
MODULE main
  VAR
    request : boolean;
    state : {ready, busy};
  ASSIGN
    init(state) := ready;
    next(state) := case
      state = ready & request
          = TRUE : busy;
      TRUE : {ready, busy};
    esac;
  SPEC AG (request -> AF state
       = busy)
```

- One "main" module
- Three sections
    - VAR: variable declaration
    - ASSIGN: variable initialization and evolution
    - SPEC: property to be verified
- Set of possible states
  {(FALSE, ready), (FALSE, busy), (TRUE, ready), (TRUE, busy)}

Variable `request` has no initial value and has no next assignment
→ `request` is an input to the system

Single process example III

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



Binary counter I

Reusable modules and expressions (module order does not matter)

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

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

Binary counter II

Loop

s	b0.ci	b0.v	b0.co	b1.ci	b1.v	b1.co	b2.ci	b2.v	b2.co
0	1	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
2	1	0	0	0	1	0	0	0	0
3	1	1	1	1	1	1	1	0	0
4	1	0	0	0	0	0	0	1	0
5	1	1	1	1	0	0	0	1	0
6	1	0	0	0	1	0	0	1	0
7	1	1	1	1	1	1	1	1	1
8	1	0	0	0	0	0	0	0	0

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

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

Binary counter III

DEFINE vs VAR

- We used `DEFINE` for `carry_out`
- The same effect is obtained by adding a variable `carry_out`
- Note we do not use `next(carry_out)`

Difference DEFINE does not require introducing a new variable, so it does not increase the state space of the FSM

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

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

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Asynchronous systems

- NuSMV allows to model asynchronous systems
- It is possible to define a collection of parallel processes, whose actions are interleaved, following an asynchronous model of concurrency
- Useful for describing communication protocols, asynchronous circuits, . . .
- Starting from NuSMV 2.5, processes are *deprecated* (one can model at a higher level, e.g. see keyword union in following slides)

Inverter ring I

Keyword process

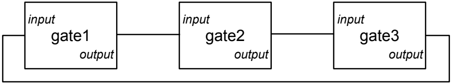
```
MODULE inverter(input)
  VAR
    output : boolean;
  ASSIGN
    init(output) := FALSE;
    next(output) := !input;

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

Inverter ring II

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

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



- Among all processes one is chosen nondeterministically: p
- p assignments are executed in parallel
- The system is not forced to eventually choose a process to execute
- In order to force a given process to execute infinitely often, we can use a fairness constraint.

Fairness constraint

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

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

- Restricts the attention of the model checker to only those execution paths along which a given formula P is true infinitely often

  LTL $M, s \models \mathbf{G}\ \mathbf{F}\ \ P$

  CTL $M, s \models \mathbf{A}\ \mathbf{G}\ (\mathbf{A}\ \mathbf{F}\ \ P)$

  CTL* $M, s \models \mathbf{A}\ \mathbf{G}\ \mathbf{F}\ \ P$

- Each process has a special variable called `running` which is `TRUE` iff that process is currently executing
- To have **G F** `running`, add the declaration to the `inverter` module

```
FAIRNESS
  running;
```

Inverter ring alternative

Modelling asynchronous processes at a higher level: keyword `union`

- Do not use processes
- Allow all gates to execute simultaneously
- Allow each gate to choose nondeterministically to re-evaluate its output or to keep the same output value (set operator `union`)
- State space size is 2^n where n is the number of gates
- But cannot require fairness

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

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

- Introduction
  - Model Checking
  - NuSMV overview
- Input language by examples
  - Synchronous systems
  - Asynchronous systems
  - Direct specification
- Simulation
  - Trace strategies
  - Interactive mode
- CTL model checking
- LTL model checking
  - Semaphore example
  - Past temporal operators
- Bounded model checking

Direct specification

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

```
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
```

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Introduction

- Model Checking
- NuSMV overview

Input language by examples

- Synchronous systems
- Asynchronous systems
- Direct specification

Simulation

- Trace strategies
- Interactive mode

CTL model checking

LTL model checking

- Semaphore example
- Past temporal operators

Bounded model checking

Trace strategies

Simulation

helps the user to explore the possible executions of the model (*traces*)

Traces can be generated

- deterministically (automatically generated by NuSMV)
- randomly (automatically generated by NuSMV)
- interactively
  - the system stops at every step, showing a list of possible future states
  - the user is requested to choose the next state
  - it is possible to specify some further constraints on next states (if such constraints are inconsistent–among each other, with future states–then the system does not accept them)

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Interactive mode

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

short.smv

- Read model with option -int
- Prepare internal structures with command go

```
mari@home$ NuSMV -int short.smv
NuSMV> go
NuSMV>
```

Interactive mode I

Choosing an initial state

Initial states can be chosen in three ways

- by default the simulator uses the *current* state, if any
- set the current state using command `goto_state`
- set the current state using command `pick_state` (use it when the current state does not exist yet: initial point or after reset)

Interactive mode II

Choosing an initial state

```
mari@home$ NuSMV -int short.smv
NuSMV> go
NuSMV> pick_state -r
NuSMV> print_current_state -v
Current state is 1.1
request = FALSE
state = ready
```

- Pick an initial state randomly

Interactive mode III

Choosing an initial state

```
NuSMV> simulate -r -k 3
********* Starting Simulation From State
    1.1 *********
NuSMV> show_traces -t
There is 1 trace currently available.
NuSMV> show_traces -v
#################### Trace number: 1
    ####################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
   request = FALSE
   state = ready
-> State: 1.2 <-
   request = TRUE
   state = busy
-> State: 1.3 <-
   request = TRUE
   state = ready
-> State: 1.4 <-
   request = TRUE
   state = busy
```

- Ask to build a three-steps simulation by picking randomly the next states of the steps
- Note that each trace has a number (1) and that each state is identified with dot notation (1.1, . . .)

Interactive mode I

Starting a new simulation

```
NuSMV> goto_state 1.4
The starting state for new trace is:
-> State 2.4 <-
 request = TRUE
 state = busy
NuSMV> simulate -r -k 3
******** Simulation Starting From State 2.4
     ********
NuSMV> show_traces 2
################### Trace number: 2
    ###################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 2.1 <-
 request = TRUE
 state = ready
...
-> State: 2.7 <-
```

- Now the user can start a new simulation by choosing a new starting state
- Extend trace 1 by first choosing state 1.4 as the current state and by then running a random simulation of length 3
- NuSMV shows all states in the trace (here represented by . . .)
- New trace is 2

Interactive mode II

Starting a new simulation

```
NuSMV> pick_state -i
*************** AVAILABLE STATES
    *************
================= State =================
0) -------------------------
    request = TRUE
    state = ready
================= State =================
1) -------------------------
    request = FALSE
    state = ready
Choose a state from the above (0-1): 1<RET>
Chosen state is: 1
```

- The user can interactively choose the states of the trace

Interactive mode III

Starting a new simulation

```
NuSMV> simulate -i -k 1
******** Simulation Starting From State 3.1
     ********
*************** AVAILABLE FUTURE STATES
    *************
================= State =================
0) -------------------------
   request = TRUE
   state = ready
================= State =================
1) -------------------------
   request = TRUE
   state = busy
================= State =================
2) -------------------------
   request = FALSE
   state = ready
================= State =================
3) -------------------------
   request = FALSE
   state = busy
Choose a state from the above (0-3): 0<RET>
Chosen state is: 0
```

- And build an interactive simulation

Interactive mode I

Specifying constraints

```
NuSMV> pick_state -c "request = TRUE" -i
*************** AVAILABLE STATES
    ***************
================ State =================
0) ------------------------
   request = TRUE
   state = ready
There's only one future state. Press Return
     to Proceed. <RET>
Chosen state is: 0
NuSMV> quit
mari@home>
```

- Specify some constraints to restrict the set of states from which the simulator will pick out
- Remark Specified constraints hold only for this step of the simulation
- Then quit the simulation

Remark Contraints specified with command `simulation` (option `-c`) hold for each step of the simulation

Introduction

- Model Checking
- NuSMV overview

Input language by examples

- Synchronous systems
- Asynchronous systems
- Direct specification

Simulation

- Trace strategies
- Interactive mode

CTL model checking

LTL model checking

- Semaphore example
- Past temporal operators

Bounded model checking

Semaphore example

Desired CTL properties

Processes proc1 and proc2 with a variable state each
States can be {idle, entering, critical, exiting}

Safety

It should never be the case that the two processes proc1 and proc2 are at the same time in the *critical state*

```
AG ! (proc1.state = critical & proc2.state = critical)
```

Liveness

If proc1 wants to enter its critical state, it eventually does

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

Semaphore example I

NuSMV source code

```
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 example II

NuSMV source code

```
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 example I

NuSMV output

```
mari@home$ NuSMV semaphore.smv
-- specification AG
-- (!(proc1.state = critical
-- & proc2.state = critical))
-- is true
-- specification AG
-- (proc1.state = entering
-- -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following
-- execution sequence
-> State: 1.1 <-
   semaphore = FALSE
   proc1.state = idle
   proc2.state = idle
-> Input: 1.2 <-
   _process_selector_ = proc1
-- Loop starts here
-> State: 1.2 <-
   proc1.state = entering
```

```
-> Input: 1.3 <-
   _process_selector_ = proc2
-> State: 1.3 <-
   proc2.state = entering
-> Input: 1.4 <-
   _process_selector_ = proc2
-> State: 1.4 <-
   semaphore = FALSE
   proc2.state = critical
-> Input: 1.5 <-
   _process_selector_ = proc1
-> State: 1.5 <-
-> Input: 1.6 <-
   _process_selector_ = proc2
-> State 1.6 <-
   proc2.state = exiting
-> Input: 1.7 <-
   _process_selector_ = proc2
-> State 1.7 <-
   semaphore = FALSE
   proc2.state = idle
```

Semaphore example II

NuSMV output

- Safety property is verified
- Liveness property is falsified, meaning that the model suffers from starvation
- Non correctness is demonstrated by a counterexample
  - Input denotes variables on which the system has no control
  - `_process_selector_` is a special variable to which is nondeterministically assigned the selected process
  - In the printout of a cyclic, infinite counter-example the starting point of the loop is marked by `-- Loop starts here`.

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- Bounded model checking

Introduction
- Model Checking
- NuSMV overview

Input language by examples
- Synchronous systems
- Asynchronous systems
- Direct specification

Simulation
- Trace strategies
- Interactive mode

CTL model checking

LTL model checking
- Semaphore example
- Past temporal operators

Bounded model checking

Semaphore example

Desired LTL properties

Processes `proc1` and `proc2` with a variable `state` each
States can be `{idle, entering, critical, exiting}`

Safety

It should never be the case that the two processes `proc1` and `proc2` are at the same time in the *critical state*

```
G ! (proc1.state = critical & proc2.state = critical)
```

Liveness

If proc1 wants to enter its critical state, it eventually does

```
G (proc1.state = entering -> F proc1.state = critical)
```

Semaphore example I

NuSMV source code

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

Semaphore example II

NuSMV source code

```
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 example

NuSMV output: the same as for CTL model checking

```
mari@home$ NuSMV semaphore.smv
-- specification AG
-- (!(proc1.state = critical
-- & proc2.state = critical))
-- is true
-- specification AG
-- (proc1.state = entering
-- -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following
-- execution sequence
-> State: 1.1 <-
   semaphore = FALSE
   proc1.state = idle
   proc2.state = idle
-> Input: 1.2 <-
   _process_selector_ = proc1
-- Loop starts here
-> State: 1.2 <-
   proc1.state = entering
```

```
-> Input: 1.3 <-
   _process_selector_ = proc2
-> State: 1.3 <-
   proc2.state = entering
-> Input: 1.4 <-
   _process_selector_ = proc2
-> State: 1.4 <-
   semaphore = FALSE
   proc2.state = critical
-> Input: 1.5 <-
   _process_selector_ = proc1
-> State: 1.5 <-
-> Input: 1.6 <-
   _process_selector_ = proc2
-> State 1.6 <-
   proc2.state = exiting
-> Input: 1.7 <-
   _process_selector_ = proc2
-> State 1.7 <-
   semaphore = FALSE
   proc2.state = idle
```

Introduction

- Model Checking
- NuSMV overview

Input language by examples

- Synchronous systems
- Asynchronous systems
- Direct specification

Simulation

- Trace strategies
- Interactive mode

CTL model checking

LTL model checking

- Semaphore example
- Past temporal operators

Bounded model checking

Past temporal operators

Past temporal operators allow to characterize properties of the path that leads to the current situation

The typical past operators are

O p (read "once p") condition p holds in one of the past time instants (past for F)

H p (read "historically p") condition p holds in all previous time instants (past for G)

p S q (read "p since q") condition p holds since a previous state where condition q holds (past for U)

Y p (read "yesterday p") condition p holds in the previous time instant (past for X)

- Introduction
    - Model Checking
    - NuSMV overview
- Input language by examples
    - Synchronous systems
    - Asynchronous systems
    - Direct specification
- Simulation
    - Trace strategies
    - Interactive mode
- CTL model checking
- LTL model checking
    - Semaphore example
    - Past temporal operators
- **Bounded model checking**

Bounded Model Checking

Checking LTL specification

```
mari@home$ NuSMV -bmc model_to_check.smv
mari@home$ NuSMV -bmc -bmc_length 4 model_to_check.smv
```

Default BMC length is 10

Finding counterexamples interactively

```
mari@home$ NuSMV -int model_to_check.smv
NuSMV> go_bmc
NuSMV> check_ltlspec_bmc_onepb -k 9 -l 0
```

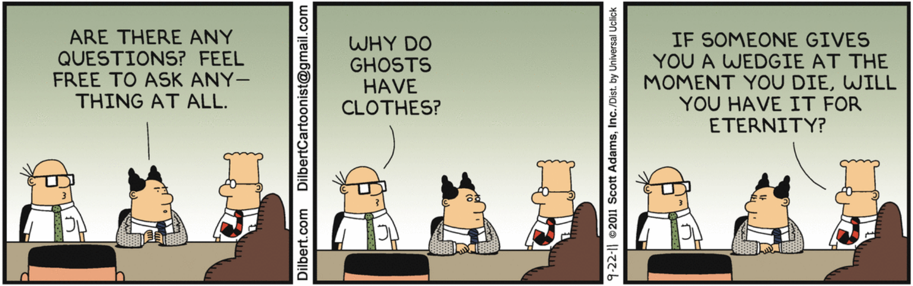
`-k 9` specifies BMC horizon (bound) equal to 9

`-l 0` is the *loopback condition* meaning that loops must start at state 0

Checking invariants

```
NuSMV> check_invar_bmc -a een-sorensson -p "y in (0..12)"
```

More details on BMC in the tutorial


ARE THERE ANY QUESTIONS? FEEL FREE TO ASK ANYTHING AT ALL.
WHY DO GHOSTS HAVE CLOTHES?
IF SOMEONE GIVES YOU A WEDGIE AT THE MOMENT YOU DIE, WILL YOU HAVE IT FOR ETERNITY?
Dilbert.com DilbertCartoonist@gmail.com
9-22-11 © 2011 Scott Adams, Inc./Dist. by Universal Uclick