An introduction to (Nu)SMV

Nuno Macedo

October 18, 2018

SMV in a nutshell

- A language for modelling *finite state machines* (FSMs)
- Support for branching and linear time *temporal logic* specifications
- Simulation and automatic verification through *model checking*, with counter-example generation

Symbolic Model Verification

- SMV language and analysis first proposed in '93 by Ken McMillan at CMU
  - Main insight: consider ranges of states rather than single states
- Several extensions throughout the years
- **NuSMV2**, an open source re-implementation from FBK
  - supports both CTL and LTL specifications
  - supports bounded SAT-based model checking
  - interactive mode and automatic verification

`http://nusmv.fbk.eu/`

Modelling: Structure

- Organized in modules, declared by **MODULE**
  - a **MODULE** `main` must always be defined
- Section **VAR** declares the state variables

```
VAR name1 : type1;
    name2 : type2;
    ...
```

- Supports simple *finite* types
- Determines the number of states in the FSM

Supported variable types

Booleans	values **TRUE** and **FALSE**, **boolean**
integers	finite ranges of integers, *n*..*m*
scalars	enumeration of symbolic values, {*a*,*b*,...}
words	bit vectors, **signed** or **unsigned** **word**[*n*]
arrays	sequences of values, possibly nested, **array** *n*..*m* **of** *type*
modules	other user defined modules

What can't be modelled?

- By definition, model checking explores every possible state, so state machine must be finite
- *State explosion* is a critical issue, so even finite states should be defined with care

Heavy chair: Modelling v0

```
MODULE main
VAR
  x  : 0..10;          -- range of integers
  y  : 0..10;          -- range of integers
  d  : {n,s,e,w};      -- enumeration of symbolic values
```

Modelling: Behaviour

Two alternative mechanisms

- Restricted syntax through assignments (**ASSIGN** section)
  - Guarantees that it is always possible to determine a next state, state machine without deadlocks
- Direct specification of state machine (**INIT**/**INVAR**/**TRANS** sections)
  - More flexible but may lead to senseless models
- Both allow non-determinism

Assignment syntax

- Parallel variable assignment in **ASSIGN** section
- Assignment to initial state and to the succeeding state, define the transition
  - **init**(*name*) := *expr1*;
  - **next**(*name*) := *expr2*;
- Alternatively, assignment to current state, define the invariant
  - *name* := *expr*;
- For each variable, either assignment of invariant or **init**/**next**

Basic expressions

- **relational**: equality =, inequality `!=`; <, >, <=, >= (for integers)
- **Boolean**: not `!`, and &, or `|`, exclusive or **`xor`**, implies `->`, iff `<->`
- **arithmetic**: `+`, `-`, $*$, integer division `/`, remainder **`mod`**
- **arrays**: access `array[n]`
- **sets**: union **`union`**, enumeration `{a,...}`, ranges `n..m`, inclusion test **`in`**
- **control flow**: conditional `guard?expr1:expr2`, cases **`case`** ... **`esac`**

Case statements

- Useful to model alternative behaviour

```
case
     guard1 : expression1;
     guard2 : expression2;
     ...
esac;
```

- Tested sequentially, the first to evaluate true is applied
- Conditions must be exhaustive, one must always evaluate true

Non-deterministic models

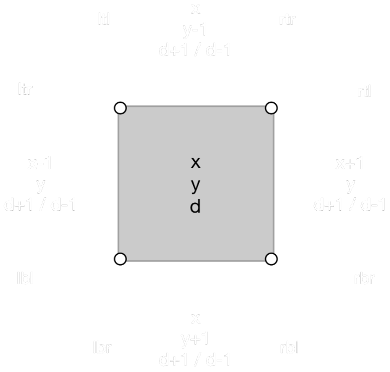
- SMV supports non-deterministic behaviour, multiple valid transitions for a state
- Achieved by
  - not providing assignments to a variable (arbitrary value in each state)
  - assign a value within a set, e.g., `next(x) := {a,b,c};`
- Useful to model the environment, out of the control of the system, or alternative / underspecified behaviour

What can't be modelled?

- Single variable assignment
- No circular dependencies
- Guarantees that the assignments are implementable and a total state machine constructed

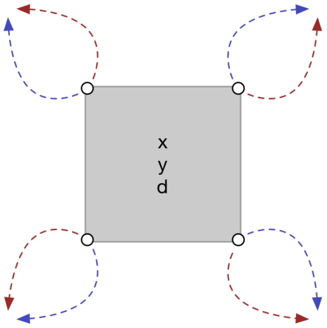
Heavy chair problem

How to model arbitrary application of actions?



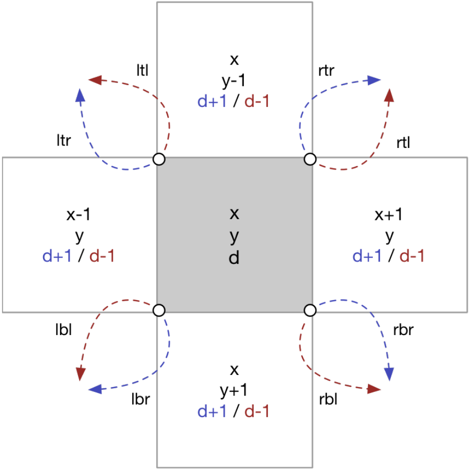
Heavy chair problem

How to model arbitrary application of actions?



Heavy chair problem

How to model arbitrary application of actions?



Heavy chair: Modelling v1

```
MODULE main
VAR
    x  : 0..5;
    y  : 0..5;
    d  : 0..3;                              -- easier to rotate

ASSIGN
    init(x) := 3;
    init(y) := 3;
    init(d) := 0;
```

Heavy chair: Modelling v1

```
MODULE main
VAR
    x  : 0..5;
    y  : 0..5;
    d  : 0..3;                                   -- easier to rotate
    op : {ltl,ltr,rtl,rtr,lbl,lbr,rbl,rbr};  -- random assignments
ASSIGN
    init(x) := 3;
    init(y) := 3;
    init(d) := 0;
```

Heavy chair: Modelling v1

```
MODULE main
VAR
    x  : 0..5;
    y  : 0..5;
    d  : 0..3;                                  -- easier to rotate
    op : {ltl,ltr,rtl,rtr,lbl,lbr,rbl,rbr};    -- random assignments
ASSIGN
    init(x) := 3;
    init(y) := 3;
    init(d) := 0;
    next(x) := case op in {ltr,lbl} : x-1;
                    op in {rtl,rbr} : x+1;
                    TRUE            : x;        -- default cases
               esac;
    next(y) := case op in {ltl,rtr} : y-1;
                    op in {lbr,rbl} : y+1;
                    TRUE            : y;
               esac;
    next(d) := case op in {rtr,rbr,ltr,lbr} : (d+1) mod 4;
                    TRUE                    : (d+3) mod 4;
               esac;
```

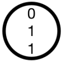
Input variables

- Environment input that is not controlled by the system is better defined through *input variables*
  - For instance, which action will be selected at each step
- Same syntax for declarations but in **IVAR** section
- Always randomly assigned, cannot be controlled by the model assignments and constraints

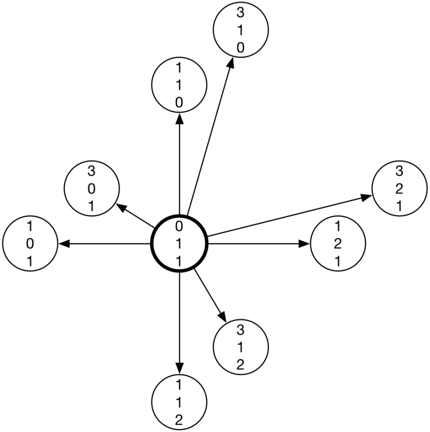
Heavy chair: Modelling v2

```
MODULE main
VAR
    x  : 0..5;
    y  : 0..5;
    d  : 0..3;                                -- easier to rotate
IVAR
    op : {ltl,ltr,rtl,rtr,lbl,lbr,rbl,rbr};  -- random assignments
ASSIGN
    init(x) := 3;
    init(y) := 3;
    init(d) := 0;
    next(x) := case op in {ltr,lbl} : x-1;
                    op in {rtl,rbr} : x+1;
                    TRUE            : x;      -- default cases
               esac;
    next(y) := case op in {ltl,rtr} : y-1;
                    op in {lbr,rbl} : y+1;
                    TRUE            : y;
               esac;
    next(d) := case op in {rtr,rbr,ltr,lbr} : (d+1) mod 4;
                    TRUE                    : (d+3) mod 4;
               esac;
```

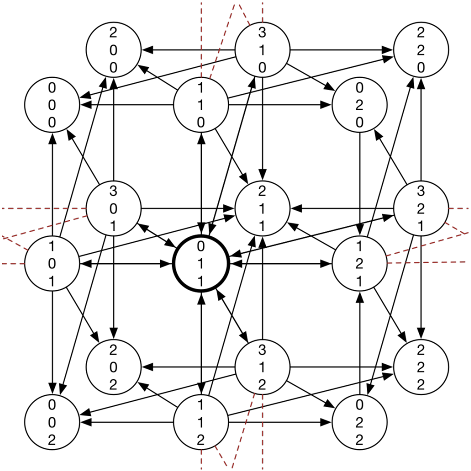
Finite heavy chair model



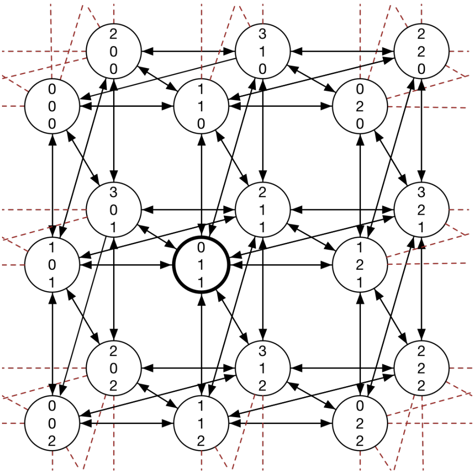
Finite heavy chair model



Finite heavy chair model



Finite heavy chair model



Finite heavy chair model

- A limit was set on the size of the board
- Operations must act within these states
- Must test whether an action is valid in each state

Macros

- Identifiers defined in a **DEFINE** section that can be re-used
- Do not generate additional variables and do not affect the model checker, simply replaced

Heavy chair: Modelling v3

```
MODULE main
VAR  x  : 0..n; y  : 0..n;                          -- parametrized size
     d  : 0..3;
IVAR op : {ltl,ltr,rtl,rtr,lbl,lbr,rbl,rbr};
DEFINE n   := 10                                    -- size of the board



ASSIGN
    init(x) := n/2; init(y) := n/2;                 -- middle of the board
    init(d) := 0;
    next(x) := case
                   op in {ltr,lbl} : x-1;
                   op in {rtl,rbr} : x+1;
                   TRUE            : x;      esac;
    next(y) := case
                   op in {ltl,rtr} : y-1;
                   op in {lbr,rbl} : y+1;
                   TRUE            : y;      esac;
    next(d) := case
                   op in {rtr,rbr,ltr,lbr} : (d+1) mod 4;
                   TRUE                    : (d+3) mod 4; esac;
```

Heavy chair: Modelling v3

```
MODULE main
VAR  x  : 0..n; y  : 0..n;                    -- parametrized size
     d  : 0..3;
IVAR op : {ltl,ltr,rtl,rtr,lbl,lbr,rbl,rbr};
DEFINE n   := 10                              -- size of the board
       inv := (x = 0 & op in {ltr,lbl}) | (x = n & op in {rtl,rbr}) |
              (y = 0 & op in {ltl,rtr}) | (y = n & op in {lbr,rbl});
                                              -- whether a valid action
ASSIGN
    init(x) := n/2; init(y) := n/2;           -- middle of the board
    init(d) := 0;
    next(x) := case inv               : x;    -- sequential tests
               op in {ltr,lbl} : x-1;         -- if stuck, do nothing
               op in {rtl,rbr} : x+1;
               TRUE            : x;     esac;
    next(y) := case inv               : y;
               op in {ltl,rtr} : y-1;
               op in {lbr,rbl} : y+1;
               TRUE            : y;     esac;
    next(d) := case inv                       : d;
               op in {rtr,rbr,ltr,lbr} : (d+1) mod 4;
               TRUE                           : (d+3) mod 4; esac;
```

Frozen variables

- Sometimes a variable has multiple possible values in the initial state but remains unchanged throughout the trace
  - For instance, the initial selection of a configuration, like the size of the board
- Same syntax for declarations but in **`FROZEN`** section
- After the initial state, cannot be controlled by the model constraints

Direct modelling

- Alternative method for modelling, define the states and transitions of the FSM directly
- Any state and transition that satisfies a predicate will belong to the FSM
- More expressive and flexible
  - Easier to group variable assignments together
- More prone to errors, harder to detect non-total transitions or empty initial states
  - If empty transition, all universal properties trivially true

Direct modelling

Defining constraints for direct modelling

INIT The initial states are exactly those that pass these constraints

INVAR The states of the machine are exactly those that pass these constraints

TRANS The transitions of the machine are exactly those whose input and output states pass these constraints

Heavy chair: Modelling v4

```
MODULE main
VAR ...
IVAR ...
DEFINE ...
INIT
  x =  n / 2 & x = y & d = 0;
TRANS
  (op = ltr -> next(x) = x-1 & next(y) = y & next(d) = (d+1) mod 4) &
  (op = lbl -> next(x) = x-1 & next(y) = y & next(d) = (d+3) mod 4) &
  (op = rtl -> next(x) = x+1 & next(y) = y & next(d) = (d+1) mod 4) &
  (op = rbr -> next(x) = x+1 & next(y) = y & next(d) = (d+3) mod 4) &
  (op = lbr -> next(x) = x & next(y) = y+1 & next(d) = (d+1) mod 4) &
  (op = rbl -> next(x) = x & next(y) = y+1 & next(d) = (d+3) mod 4) &
  (op = ltl -> next(x) = x & next(y) = y-1 & next(d) = (d+1) mod 4) &
  (op = rtr -> next(x) = x & next(y) = y-1 & next(d) = (d+3) mod 4) &
  !inv
```

Modelling software systems

- Besides the program variables, the model must also encode which statement is to be executed next
- This is usually encoded by an additional variable that denotes the location, or the *program counter*, of the execution
- Input variables (**IVAR**) can be used to model the *process scheduler* of the operating system

Peterson's mutual exclusion algorithm

Shared state

```
bool flag[2] = {false, false};
int turn;
```

Process 0

```
idle: flag[0] = true;
want: turn = 1;
wait: while (flag[1] && turn == 1)
      { /* busy wait */ }
crit: // critical section
      flag[0] = false;
```

Process 1

```
idle: flag[1] = true;
want: turn = 0;
wait: while (flag[0] && turn == 0)
      { /* busy wait */ }
crit: // critical section
      flag[1] = false;
```

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Peterson's algorithm: Modelling v1

```
MODULE main
VAR
  flg : array 0..1 of boolean;              // program variables
  trn : 0..1;                               // program variables

IVAR
  run : 0..1;                               // process scheduler
ASSIGN
  next(trn) :=


  init(pc[0]) := idle;
  next(pc[0]) :=





  init(flg[0]) := FALSE;
  next(flg[0]) :=



  ...
```

Peterson's algorithm: Modelling v1

```
MODULE main
VAR
  flg : array 0..1 of boolean;                    // program variables
  trn : 0..1;                                     // program variables
  pc  : array 0..1 of {idle,want,wait,crit}; // program counter
IVAR
  run : 0..1;                                     // process scheduler
ASSIGN
  next(trn) := case run=0 & pc[0]=want: 1;
                    run=1 & pc[1]=want: 0;
                    TRUE              : trn; esac;
  init(pc[0]) := idle;
  next(pc[0]) := case run=0 & pc[0]=idle                    : want;
                      run=0 & pc[0]=want                    : wait;
                      run=0 & pc[0]=wait & !(flg[1] & trn=1): crit;
                      run=0 & pc[0]=crit                    : idle;
                      TRUE                                  : pc[0]; esac;
  init(flg[0]) := FALSE;
  next(flg[0]) := case run=0 & pc[0]=idle: TRUE;
                       run=0 & pc[0]=crit: FALSE;
                       TRUE              : flg[0]; esac;
  ...
```

Modules

- SMV supports modularized and hierarchical systems
- A defined module may be instantiated multiple times inside another one
- Parameters are passed by *reference*, either to complete modules or variables
  - reference to the current module passed by **`self`**
  - variables inside modules accessed by `.`
- The composition is synchronous
  - assignments in all modules are executed at once, a step of the system is a step on every module

Peterson's algorithm: Modelling v2

```
MODULE proc(id,alt,m) // id, other process flag, the main scheduler
VAR flg : boolean;
    pc  : {idle,want,wait,crit};
ASSIGN
  init(pc) := idle;
  next(pc) := case ...
                 m.run=id & pc=wait & !(alt & m.trn!=id): crit;
                 ... esac;
  init(flg) := FALSE;
  next(flg) := case m.run=id & pc=idle: TRUE;
                    m.run=id & pc=crit: FALSE;
                    TRUE                : flg; esac;
```

Peterson's algorithm: Modelling v2

```
MODULE proc(id,alt,m) // id, other process flag, the main scheduler
VAR flg : boolean;
    pc  : {idle,want,wait,crit};
ASSIGN
  init(pc) := idle;
  next(pc) := case ...
                 m.run=id & pc=wait & !(alt & m.trn!=id): crit;
                 ... esac;
  init(flg) := FALSE;
  next(flg) := case m.run=id & pc=idle: TRUE;
                    m.run=id & pc=crit: FALSE;
                    TRUE              : flg; esac;

MODULE main
VAR trn  : 0..1;
    p0   : proc(0,p1.flg,self);
    p1   : proc(1,p0.flg,self);
IVAR run : 0..1;
ASSIGN
  next(trn) := case run=0 & p0.pc=want: 1;
                    run=1 & p1.pc=want: 0;
                    TRUE              : trn; esac;
```

Simulation

- Models can be interactively simulated in NuSMV
- States are iteratively chosen (randomly or by the user) according to the defined model
- Multiple traces may be generated in the same session
  - State $m.n$ means step n at trace m

Minimal simulation example

Simulation run

```
$ NuSMV -int chair.smv     -- start interactive mode
NuSMV> go                  -- process the model
NuSMV> pick_state -v       -- pick an initial state
NuSMV> simulate -k 2 -v    -- advance two steps
NuSMV> show_trace          -- print the trace
```

By default, unchanged variables are omitted

Minimal simulation example

Simulation output

```
    <!-- ################### Trace number: 1 ################### -->
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    x = 5
    y = 5
    d = 0
    m = 0
    n = 10
    inv = FALSE
  -> Input: 1.2 <-
    op = ltl
  -> State: 1.2 <-
    y = 4
    d = 1
  -> Input: 1.3 <-
    op = rtr
  -> State: 1.3 <-
    y = 3
    d = 0
```

Useful simulation commands

`$ NuSMV -int`	Start NuSMV in interactive mode
`go`	Read the model and initialize the system for verification
`show_vars`	Show the state variables and their types
`reset`	Reset the process when the file changed
`quit`	Quit NuSMV

Useful simulation commands

`pick_state` Select an initial state

- `-i` Ask the user to select the state from a list
- `-v` Print the selected state and variables

`simulate` Generate a sequence of states from the current

- `-i` Ask the user to select the steps from a list
- `-v` Print the selected states and variables
- `-k` The number of steps to be generated

`print_current_state` Prints the name of the current state

- `-v` Print the selected states and variables

`show_traces` Prints the generated traces

- `-v` Print the state variables

Specification

- Support for both LTL (**`LTLSPEC`**) and CTL (**`CTLSPEC`**) specifications
- The model checker can automatically checker whether it holds
  - From the command-line: `NuSMV chair.smv`
  - In interactive mode: `check_ltlspec` or `check_ctlspec`

CTL

- Supported CTL operators:

EX f	there exists a path where f holds in the succeeding state
EG f	there exists a path where f always holds
EF f	there exists a path where f eventually holds
AX f	in all paths f holds in the succeeding state
AG f	in all paths f always holds
AF f	in all paths f eventually holds
E[f **U** g]	there exists a path where f holds until g does
A[f **U** g]	in all paths f holds until g does

LTL

- Supported LTL operators (including past-time):

	X	f	f holds in the succeeding state
	G	f	f always holds
	F	f	f eventually holds
f	**U**	g	f holds until g does
f	**V**	g	g always holds or until f does
	Y	f	f held in the previous state
	H	f	f always held in the past
	O	f	f once held in the past
f	**S**	g	f held since g did
f	**T**	g	g always held or since g did

Heavy chair: Specification

- Back to the heavy chair puzzle
  - **G** `(x = n/2 & y = (n/2)+1 & d = 0)?`
  - **G** `! (x = n/2 & y = (n/2)+1 & d = 0)?`
  - **F** `(x = n/2 & y = (n/2)+1 & d = 0)?`
  - **F** `! (x = n/2 & y = (n/2)+1 & d = 0)?`

  - **AG** `(x = n/2 & y = (n/2)+1 & d = 0)?`
  - **EG** `(x = n/2 & y = (n/2)+1 & d = 0)?`
  - **AF** `(x = n/2 & y = (n/2)+1 & d = 0)?`
  - **EF** `(x = n/2 & y = (n/2)+1 & d = 0)?`

Peterson's algorithm: Specification

- Back to the heavy chair puzzle
  - **G** !(pc[0]=crit & pc[1]=crit)?
  - pc[0]=want -> **F** pc[0]=crit?
  - **G** (pc[0]=want -> **F** pc[0]=crit)?

  - **AG** !(pc[0]=crit & pc[1]=crit)?
  - **AG** (pc[0]=want -> **EF** pc[0]=crit)?
  - **AG** (pc[0]=want -> **AF** pc[0]=crit)?

Fairness

- Some systems are only correct if a certain realistic *fairness* conditions are met
  - For instance, the scheduler will not prioritize the same process indefinitely
- Can be encoded in LTL but not CTL
- NuSMV provides special **`JUSTICE`** `f` constraints
  - Formula `f` will be true infinitely often in all fair paths

Peterson's algorithm: Modelling v1

```
MODULE main
VAR
  flg : array 0..1 of boolean;                   // program variables
  trn : 0..1;                                    // program variables
  pc  : array 0..1 of {idle,want,wait,crit}; // program counter
IVAR
  run : 0..1;                                    // process scheduler
ASSIGN
  ...
  ...
LTLSPEC G (pc[0]=want -> F pc[0]=crit)
```

Peterson's algorithm: Modelling v1

```
MODULE main
VAR
  flg : array 0..1 of boolean;                  // program variables
  trn : 0..1;                                   // program variables
  pc  : array 0..1 of {idle,want,wait,crit}; // program counter
IVAR
  run : 0..1;                                   // process scheduler
ASSIGN
  ...
  ...
LTLSPEC G (pc[0]=want -> F pc[0]=crit)
JUSTICE run=1
JUSTICE run=2
```

Verification

- The model checker can automatically checker whether it holds
  - From the command-line: `NuSMV chair.smv`
  - In interactive mode: `check_ltlspec` or `check_ctlspec`

Counter-examples traces

- The model checker attempts to verify the property and present a counter-example otherwise
- Counter examples to **F**/**AG** properties must be infinite; a trace with a loop is returned
- Traces are not necessarily minimal (LTL checking in particular requires looping traces)
- Counter-examples to existential properties **E** cannot be shown, as would entail presenting all traces (just the initial states)

Heavy chair: Verification

```
LTLSPEC F (x = n / 2 & y = n / 2 & d = 0)
```

Counter-example reported

```
-- specification  F ((x = n / 2 & y = n / 2) & d = 0)  is true
```

Heavy chair: Verification

```
LTLSPEC F (x = n / 2 & y = n / 2 & d = 1)
```

Counter-example reported

```
-- specification  F ((x = n / 2 & y = n / 2) & d = 1)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    x = 5
    y = 5
    d = 0
  -> Input: 1.2 <-
    op = ltl
  -> State: 1.2 <-
    y = 4
    d = 1
  -> Input: 1.3 <-
    op = rbl
  -> State: 1.3 <-
    y = 5
    d = 0
```

Model finding

- This mechanism can also be used to search for solutions to problems, by asking to falsify their inverse
- For instance, if `state` is reachable:

  G `!state`? no, here's a witness leading to `state`

Heavy chair: Model finding

LTLSPEC G !(x = n/2 & y = n/2 & d = 2)

Witness to x = n/2 & y = n/2 & d = 2

```
-- specification  G !((x = n / 2 & y = n / 2) & d = 2)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    x = 5
    y = 5
    d = 0
  -> Input: 1.2 <-
    op = ltl
  -> State: 1.2 <-
    y = 4
    d = 1
    ...
  -> Input: 1.6 <-
    op = rbl
  -> State: 1.6 <-
    y = 4
    d = 1
```

Useful links

- NuSMV Homepage.
  `http://nusmv.fbk.eu/`
- NuSMV Tutorial.
  `http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf`
- NuSMV User Manual.
  `http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`