An introduction to (Nu)SMV

Part I: Modelling and Simulation

Nuno Macedo

September 27, 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 model 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

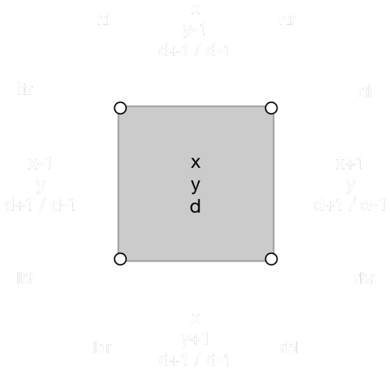
- SMV supports non-deterministic behaviour, multiple valid transitions for a state
- Achieved by
  - not providing assignments to a variable (arbitrary value in each state)
  - assign a value within a set, e.g., `next(x) := {a,b,c};`
- Useful to model the environment, out of the control of the system,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

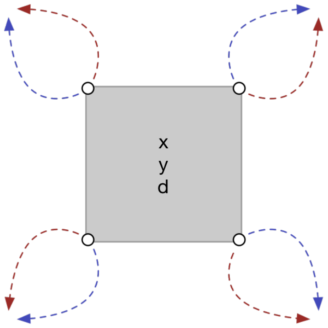
Heavy chair problem

How to model arbitrary application of actions?



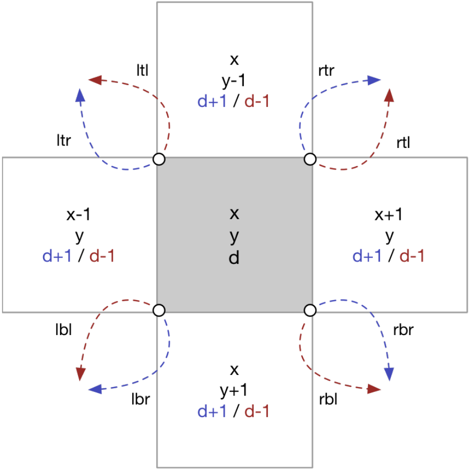
Heavy chair problem

How to model arbitrary application of actions?



Heavy chair problem

How to model arbitrary application of actions?



Heavy chair model 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 model 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 model 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;
```

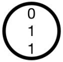
Input variables

- 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

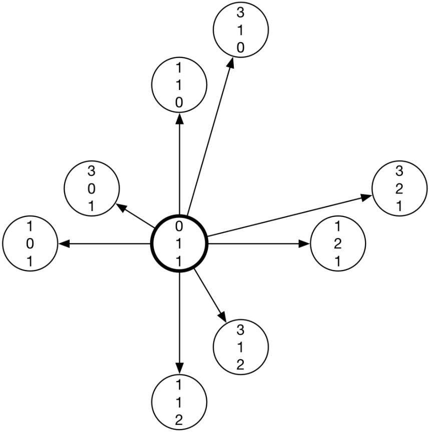
Heavy chair model v2

```
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;
```

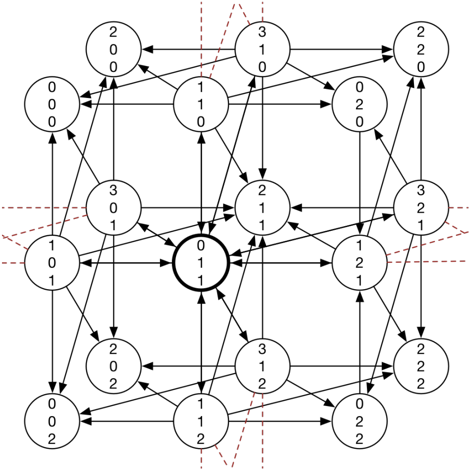
Finite heavy chair model



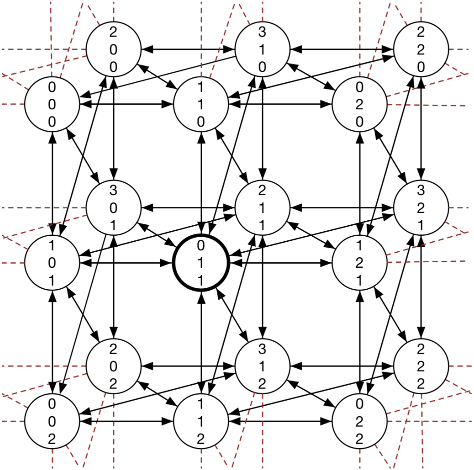
Finite heavy chair model



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 model 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;          -- if stuck, do nothing
                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 model 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 model v4

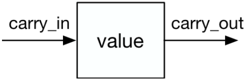
```
MODULE main
VAR ...
IVAR ...
DEFINE ...
INIT
  x =  n / 2 & x = y & d = 0;
TRANS
  (op = ltr -> next(x) = x-1 & next(y) = y & next(d) = (d+1) mod 4) &
  (op = lbl -> next(x) = x-1 & next(y) = y & next(d) = (d+3) mod 4) &
  (op = rtl -> next(x) = x+1 & next(y) = y & next(d) = (d+1) mod 4) &
  (op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

Modules

- SMV supports modularized and hierarchical systems
- A defined module may be instantiated multiple times inside another one
- Parameters are passed by reference
- The composition is synchronous
    - assignments in all modules are executed at once, a step of the system is a step on every model

3-bit counter (from the NuSMV tutorial)

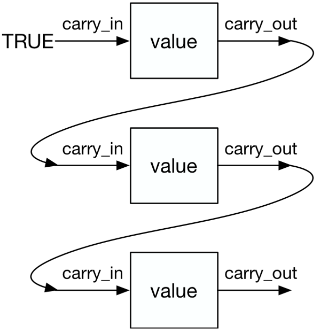
```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;
```



3-bit counter (from the NuSMV tutorial)

```
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);
```



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

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

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 and verification at a glance

- Support for both LTL (**LTLSPEC**) and CTL (**CTLSPEC**)
- Basic LTL operators:

  X the property must hold in the next state
  G the property must hold in every state
  F the property must eventually hold in a

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

Specification and verification at a glance

- 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)?`

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`

Two river crossing puzzles

- Fox, goose and bag of beans puzzle.
  en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle
- Bridge and torch problem.
  en.wikipedia.org/wiki/Bridge_and_torch_problem