

# Model checking

Cinzia Bernardeschi

Department of Information Engineering

University of Pisa

FMSS, 2019-2020

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

# 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
- 
- DECLARATION of variables
  - ASSIGNMENTS that define the initial states
  - ASSIGNMENTS that define the transition relation

# Data types

The language provides the following types

- ▶ booleans
- ▶ enumerations (cannot contain any boolean value (FALSE, TRUE))
- ▶ 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  
array 0..3 of boolean  
array 10..20 of OK, y, z
- ▶ .....

DEFINE is used for abbreviations

DEFINE <id> := <simple\_expression> ; no constraint on order such as in C where a declaration of a variable should always be placed in text above the variable

# 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;  
...  
TRUE: exprN;  
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).

next operator cannot be applied twice, i.e. next(next(a))

# Finite state machine-FSM

## ▶ Variables

- ▶ state variables
- ▶ input variables
- ▶ frozen variables

variables that retain their initial value throughout the evolution of the state machine

- ▶ transition relation describing how inputs leads from one state to possibly many different states
- ▶ Fairness conditions that describe constraints on the valid paths of the execution of the FSM

FMS = finite transition system

# Finite Transition system

- ▶ Initial state:  
init(<variable>) := <simple\_expression> ;  
variables not initialised 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



## More on variables

- ▶ state variables (VAR)
- ▶ input variables (IVAR)
  - are used to label transitions of the Finite State Machine.
  - input variables cannot occur in left-side of assignments
  - IVAR  $i$  : boolean;
  - ASSIGN  $\text{init}(i) := \text{TRUE};$  – legal
  - $\text{next}(i) := \text{FALSE};$  – illegal
- ▶ frozen variables (FROZENVAR)
  - variables that retain their initial value throughout the evolution of the state machine
  - ASSIGN
  - $\text{init}(a) := d;$  – legal
  - $\text{next}(a) := d;$  – illegal

# Constraints

- ▶ INIT constraint

The set of initial states of the model is determined by a boolean expression under the INIT keyword.

- ▶ INVAR constraint

The set of invariant states can be specified using a boolean expression under the INVAR key- word.

- ▶ TRANS constraint

The transition relation of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a boolean expression, introduced by the TRANS keyword.

# Constraints

## ► ASSIGN Constraint

ASSIGN  $a := \text{exp}$ ;

ASSIGN  $\text{init}(a) := \text{exp}$

ASSIGN  $\text{next}(a) := \text{exp}$

Assignments describe a system of equations that say how the FSM evolves through time.

# Constraints

## ► FAIRNESS Constraints

A fairness constraint restricts to fair execution paths. Paths that satisfy the expression `simple_expr` below, which is assumed to be boolean.

When evaluating formulae, the model checker considers path quantifiers to apply only to fair paths.

FAIRNESS `simple_expr` ;

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

# MODULE instantiation

An instance of a module is created using the VAR declaration. In the declaration actual parameters are specified

In the following example, the semantic of module instantiation is similar to call-by-reference (the variable a below is assigned the value TRUE)

```
MODULE main
```

```
VAR
```

```
a : boolean;
```

```
b : foo(a);
```

```
...
```

```
MODULE foo(x)
```

```
ASSIGN
```

```
x := TRUE;
```

# MODULE instantiation

in the following example, the semantic of module instantiation is similar to call-by-value

```
MODULE main
```

```
...
```

```
DEFINE
```

```
a := 0;
```

```
VAR
```

```
b : bar(a);      b is a module of type bar declared inside module main
```

```
...
```

```
MODULE bar(x)
```

```
DEFINE
```

```
a := 1;
```

```
y := x;
```

The value of y is 0

# Composition of modules

```
MODULE mod  
VAR  
out: 0..9;  
ASSIGN  
next(out) := (out + 1) mod 10;
```

```
MODULE main  
VAR  
m1 : mod;  
m2 : mod;  
sum: 0..18;  
ASSIGN sum := m1.out + m2.out;
```

. used to access the components of modules (e.g., variables)

**self** used for the current module



# Composition of modules

Module declarations may be parametric.

```
MODULE mod(in)
```

```
VAR out: 0..9;
```

```
...
```

```
MODULE main
```

```
VAR
```

```
m1: mod(m2.out);
```

```
m2 : mod(m1.out);
```

```
...
```

# Composition of modules

- ▶ modules have parameters (input/output parameters)
- ▶ variables declared in a module are local to the module
- ▶ synchronous composition: all modules move at each step (by default)
- ▶ asynchronous composition (modules instantiated with the keyword **process**): one process moves at each step (it is possible to define a collection of parallel processes, whose actions are interleaved, following an asynchronous model of concurrency)

One process is non-deterministically chosen, and the assignment statements declared in that process are executed in parallel. Variables not assigned by the process remains unchanged. Next process to execute is chosen non-deterministically.

## Example

```
MODULE main
VAR
semaphore : boolean;
proc1: process user(semaphore);
proc2: process user(semaphore);
.....
```

**running**: a special variable of each process - TRUE if and only if that process is currently executing. It can be used in a fairness constraint (formula true infinitely often).

## NuSMV 2.6 documents

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 (/share/nusmv/doc/nusmv.pdf)

example available also at the URL

<<http://nusmv.fbk.eu/examples/examples.html>>

Examples below are taken from the tutorial.

## 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 = TRUE : busy;
    TRUE: {ready, busy };
esac;
```

# A simple program

Build the transition system (also named Finite state machine - FSM)

# Running NuSMV

`./NuSMV -int`

activates an interactive shell

`read_model [-i filename]`

reads the input model

`go`

reads and initializes NuSMV for verification or simulation

`reset`

resets the whole system

`help`

shows the list of all commands

`quit`

stops the program

# Simulation

```
simulate [-p | -v] [-r | -i] -k steps
```

generates a sequence of at most steps states 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



# Simulation

`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

```
go
pick_state -r
print_current_state -v
simulate -r -k 3
show_traces -t
show_traces -v

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

# Verification

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:

NuSMV file.smv

## An example

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

## Exercise

### Mutual Exclusion

Implement mutual exclusion between two processes, using a boolean variable semaphore.

Each process has four states: idle, entering, critical and exiting.

The entering state indicates that the process wants to enter its critical region.

If the variable semaphore is FALSE, it goes to the critical state, and sets semaphore to TRUE.

On exiting its critical region, the process sets semaphore to FALSE again.

## Exercise

MODULE main

VAR

semaphore : boolean;

proc1: process user(semaphore);

proc2: process user(semaphore);

ASSIGN

init(semaphore) := FALSE;

MODULE user(semaphore)

VAR

state : {idle, entering, critical, exiting};

.....

```
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
```

# Exercise

## Properties

1. It never is the case that the two processes `proc1` and `proc2` are at the same time in the critical state

$AG \neg (proc1.state = critical \ \& \ proc2.state = critical)$

2. if `proc1` wants to enter its critical state, it eventually does - a liveness property

$AG (proc1.state = entering \rightarrow AF \ proc1.state = critical)$

Counter-example path. It can happen that `proc1` never enters its critical region.