

Tool demonstration: SMV

The SMV system

SMV was created by [Ken McMillan](#) (1992)

It is a model checker for **CTL**.

Useful for modelling **finite-state systems**, especially **synchronous** or **asynchronous** circuits.

Historically, SMV was the first tool to enable the verification of large hardware systems (by using **BDDs**).

Where to find SMV?

In the World-Wide Web:

`http://www-2.cs.cmu.edu/~modelcheck/smv.html`

(including extensive documentation)

The CTL Problem

Given a Kripke Structure \mathcal{K} and a CTL formula ϕ ,
do (all) initial states of \mathcal{K} satisfy ϕ ?

In SMV a Kripke structure consists of modules and processes,
the state space of each process consists of variables.

Transitions are given by declaring 'next' values for each variable.

Atomic propositions are expressions over the variables, their interpretation is
done automatically in the 'natural' way.

Technical note: In SMV, multiple initial states are possible.

Syntax example (1/3)

```
-- This is a comment.
```

```
MODULE main
```

```
VAR
```

```
    x : boolean;
```

```
    y : {q1, q2};
```

```
-- continued on the next slide
```

Remarks:

- possible variable types: boolean, integers, enumeration types
- all variable types must be **finite**

Syntax example (2/3)

ASSIGN

```
init(x) := 1;           -- Initial value
next(x) := case        -- Transition relation for x
    x: 0;
    !x: 1;
esac;

next(y) := case        -- Transition relation for y
    x & y=q1: q2;
    x & y=q2: {q1, q2}; -- non-determinism
    1      : y;
esac;
```

SMV Example: Remarks

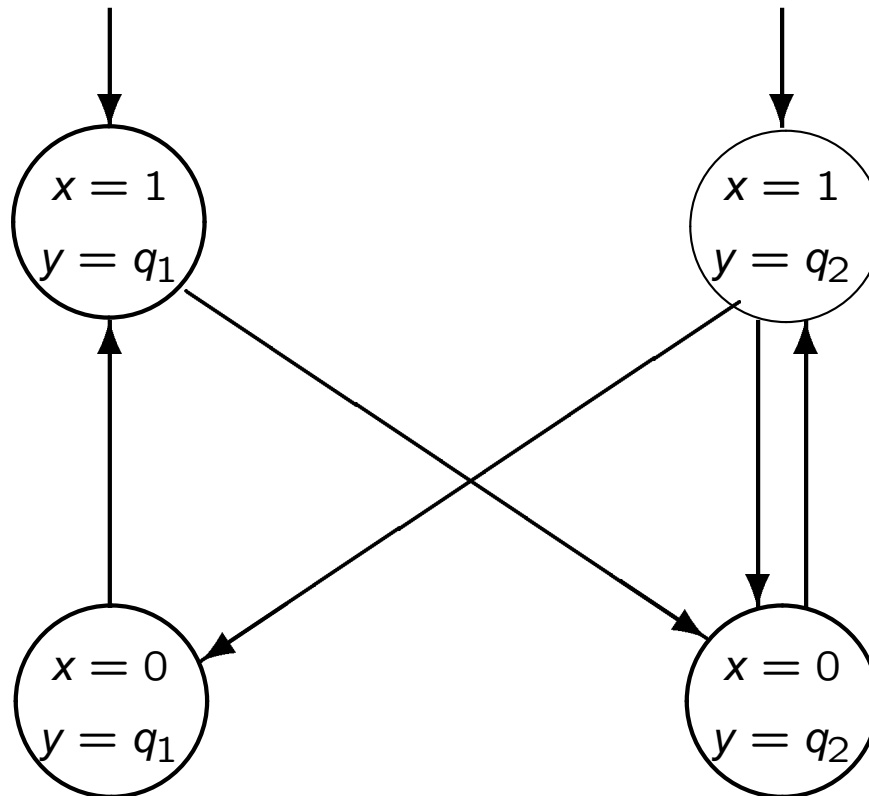
Initial states are given by the `init` predicates; uninitialized variables can take on any initial value.

Transitions given by `synchronous` composition of the `next` predicates.

`case` expressions are evaluated from top to bottom; the first branch evaluating to true is taken.

Non-determinism is possible by listing a set of values.

The resulting transition system



Syntax example (3/3)

-- Is q1 always reachable from q2?

```
SPEC AG (y=q2 -> EF y=q1)
```

-- Is x true infinitely often on all paths?

```
SPEC AG AF x
```

Remarks:

- Multiple formulas can be given in one model, and SMV will check them all.

Demonstration: xy.smv

Using modules (1/2)

Modules may be **parametrised**. Example:

```
MODULE counter_cell(carry_in)
VAR
    value : boolean;
ASSIGN
    init(value) := 0;
    next(value) := (value + carry_in) mod 2;
DEFINE
    carry_out := value & carry_in;
```

Remarks:

- This module is parametrised by `carry_in`.
- `DEFINE` is a 'macro'

Using modules (2/2)

Modules are **instantiated** like variable definitions:

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
```

This system acts like a 3-bit counter, starting at 0, counting to 7, and then starting over.

Remark: A module called `main` must always be present, and it is this module whose specifications are evaluated.

Demonstration: counter.smv

Asynchronous Systems

Up to now all examples worked **synchronously**, i.e. all variables und modules make transitions **at the same time**.

When a module is instantiated with the keyword **process** (see example), then the system *non-deterministically* chooses a process in each step and carries out its transitions. All other processes do nothing in that step. This mode of execution is called **asynchronous**.

Technical note: The system may also choose to execute *no process* in a step (i.e. the system may *stutter*).

Example: Mutex

var turn : {0,1};

while true do

q_0 non-critical section

q_1 **await** (turn=0);

q_2 critical section

q_3 turn:=1;

od

while true do

r_0 non-critical section

r_1 **await** (turn=1);

r_2 critical section

r_3 turn:=0;

od

Mutex in SMV (1/2)

```
MODULE main
```

```
VAR
```

```
    turn: boolean;
```

```
    p0: process p(0,turn);
```

```
    p1: process p(1,turn);
```

```
SPEC
```

```
    AG !(p0.state = critical & p1.state = critical)
```

Mutex in SMV (2/2)

```
MODULE p (nr,turn)
VAR
  state: {non_critical, critical};
ASSIGN
  init(state) := non_critical;
  next(state) := case
    state = non_critical & turn != nr: non_critical;
    state = non_critical & turn = nr : critical;
    state = critical: {critical,non_critical};
  esac;
  next(turn) := case
    state = critical & next(state) = non_critical: !nr;
    1 : turn;
  esac;
```

Fairness

In the Mutex example, let us check the following property instead:

```
SPEC
```

```
AG (p0.state = non_critical -> AF p0.state = critical)
```

Spin reports an error because of infinite stuttering. We can exclude stuttering executions using fairness constraints. Fairness constraints can be added with the **FAIRNESS** keywords, like this:

```
FAIRNESS
```

```
p0.running & p1.running
```

Now, only executions that satisfy the fairness condition infinitely often are considered. The special variable `running` is true after every step that the respective process has taken. With the additional fairness constraint, the verification now succeeds.