Tool demonstration: SMV
The SMV system

SMV was designed by Ken McMillan (CMU, 1992)

Model-checking tool for CTL (with fairness)

Useful for describing finite structures, especially synchronous or asynchronous circuits.

SMV was the first tool suitable for verifying large hardware systems (by employing BDDs).
Information/downloading SMV

In the world-wide web:

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

(includes extensive manual)
The problem: Given $\mathcal{K}$ (with multiple initial states) and $\phi$, do all initial states of $\mathcal{K}$ satisfy $\phi$?

In SMV, structures consist of “modules” and “processes”, that manipulate a number of variables.

Transitions are specified by declaring their ‘next’ value, depending on their current value.

Atomic propositions may talk about variables and are interpreted ‘naturally’ (as in Spin).
-- This is a comment.

MODULE main
VAR
  x : boolean;
  y : {q1,q2};

-- to be continued

Remarks:

- Data types: boolean, integer, enumerations
- all variable with \textit{finite} range
ASSIGN

init(x) := 1; -- initial value

next(x) := case -- transition relation
    x: 0;
    !x: 1;
esac;

next(y) := case
    x & y=q1: q2;
    x & y=q2: \{q1,q2\}; -- non-determinism
    1 : y;
esac;
Syntax example: Remarks

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

The transitions of the system the synchronous composition of the next predicates.

case expressions are evaluated top to bottom, the first case that fits is taken.

Non-determinism can be introduced by giving multiple successor values.
The resulting structure

\[
\begin{align*}
    x &= 1 \\
    y &= q_1 \\
    x &= 0 \\
    y &= q_1
\end{align*}
\]
-- Is q1 always reachable from q2?
SPEC AG (y=q2 -> EF y=q1)

-- Is x true infinitely often in every execution?
SPEC AG AF x

Remarks:

- One can give multiple CTL formulae, SMV will check them all one by one.

Demonstration: xy.smv
Modules (1/2)

Modules may be parametrized. Example:

```plaintext
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;

Remark:

- The parameter of this module is `carry_in`.
- `DEFINE` declares a ‘macro’.
```
Modules may be instantiated like variables:

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

This system behaves like a three-bit counter, i.e. “counts” from 0 to 7 and then resets.

Remark: There must be one module named main, and SMV will evaluate the specifications of this module.

Demonstration: counter.smv
Asynchronous systems

All examples up to now were synchronous, i.e. all variables and modules take a transition at the same time.

When modules are instantiated with the keyword process (see next example), then in each step one process makes a step while the others do nothing (asynchronous composition, interleaving).

Alternatively, no process may make a step (“stuttering”).
Example: Mutex

```plaintext
var turn : \{0,1\};

while true do
  q₀ non-critical section
  q₁ await (turn=0);
  q₂ critical section
  q₃ turn:=1;
  od

while true do
  r₀ non-critical section
  r₁ await (turn=1);
  r₂ critical section
  r₃ turn:=0;
  od
```
MODULE main
VAR
    turn: boolean;
    p0: process p(0,turn);
    p1: process p(1,turn);

SPEC
    AG !(p0.state = critical & p1.state = critical)
MODULE p (nr, turn)
VAR
    state: {nonCriticalSection, critical};
ASSIGN
    init(state) := nonCriticalSection;
    next(state) := case
        state = nonCriticalSection & turn != nr: nonCriticalSection;
        state = nonCriticalSection & turn = nr : critical;
        state = critical: {critical, nonCriticalSection};
    esac;
    next(turn) := case
        state = critical & next(state) = nonCriticalSection: !nr;
        1 : turn;
    esac;
In the mutex example, the following specification is evaluated to false.

\[
\text{SPEC} \quad \text{AG (p0.state = non\_critical} \rightarrow \text{AF p0.state = critical)}
\]

This is because SMV allows the system to “stutter” forever (i.e. to do nothing). One can exclude such behaviours using the keyword \text{FAIRNESS}, e.g. as follows:

\[
\text{FAIRNESS} \quad \text{p0.running} \& \text{p1.running}
\]

The internal variable \text{running} becomes true whenever the corresponding process makes a step. With this addition, the verification succeeds.