The NuSMV system

(Symbolic Model Verifier)

uses BDDs
(binary decision diagrams)

and SAT (new)
→ for bounded model checking

prompt > NuSMV file.smv

<u>Input</u>: program describing a model
+
specifications (LTL, CTL formulas)

<u>Output</u>: "<u>true</u>" if spec holds for all initial states

or
a trace showing why the spec is false in the mod

An SMV program may consist of several modules, one of them must be "main".

Variables: boolean, enumeration type, integer sub range, array, ...

Values of variables in each state are defined using init, next.

- the main module has no formal parameters
- transitions to states can be chosen non-deterministically.
- Comment: anything starting with -- and ending with a new line.

SPEC

!a & !b &

AG (a&b -> AX(b)) &

AG (!a&b -> AX(a & !b)) &

AG (!a&!b -> AX(a&!b))

Example

Notations

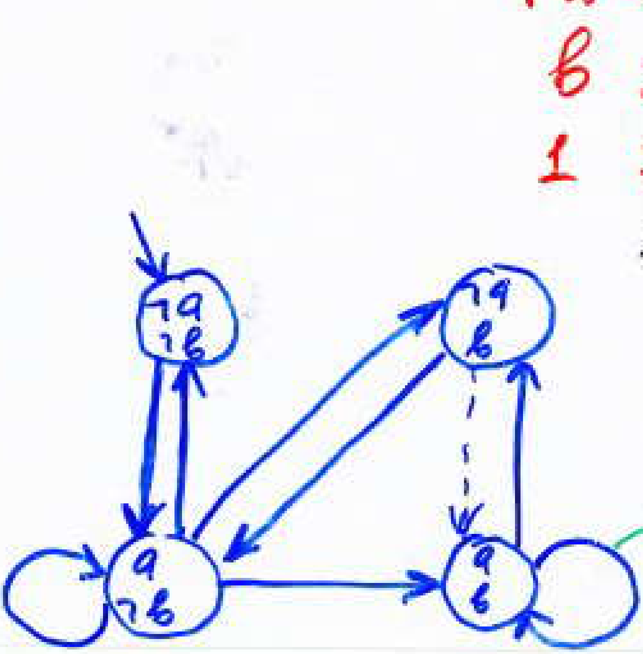
- $\neg$ = `!`
- $\wedge$ = `&`
- $\vee$ = `|`
- $\rightarrow$ = `->`
- $\leftrightarrow$ = `<->`
- true = 1
- false = 0

```
MODULE : main
VAR a : boolean;
    b : boolean;
ASSIGN
    init(a) := 0;
    init(b) := 0;
    next(a) := case
        !a : 1;
        1 : {0,1};
    esac;
    next(b) := case
        !a : 0;
        b : 1;
        1 : {0,1};
    esac;
```

the value of a in the next state

o.w. the "default" case either 0 or 1

If several cases are true then the first from the top is execute



a non-deterministic model

status : (n, t, c) of the other process

instantiations of prc

main

CTL

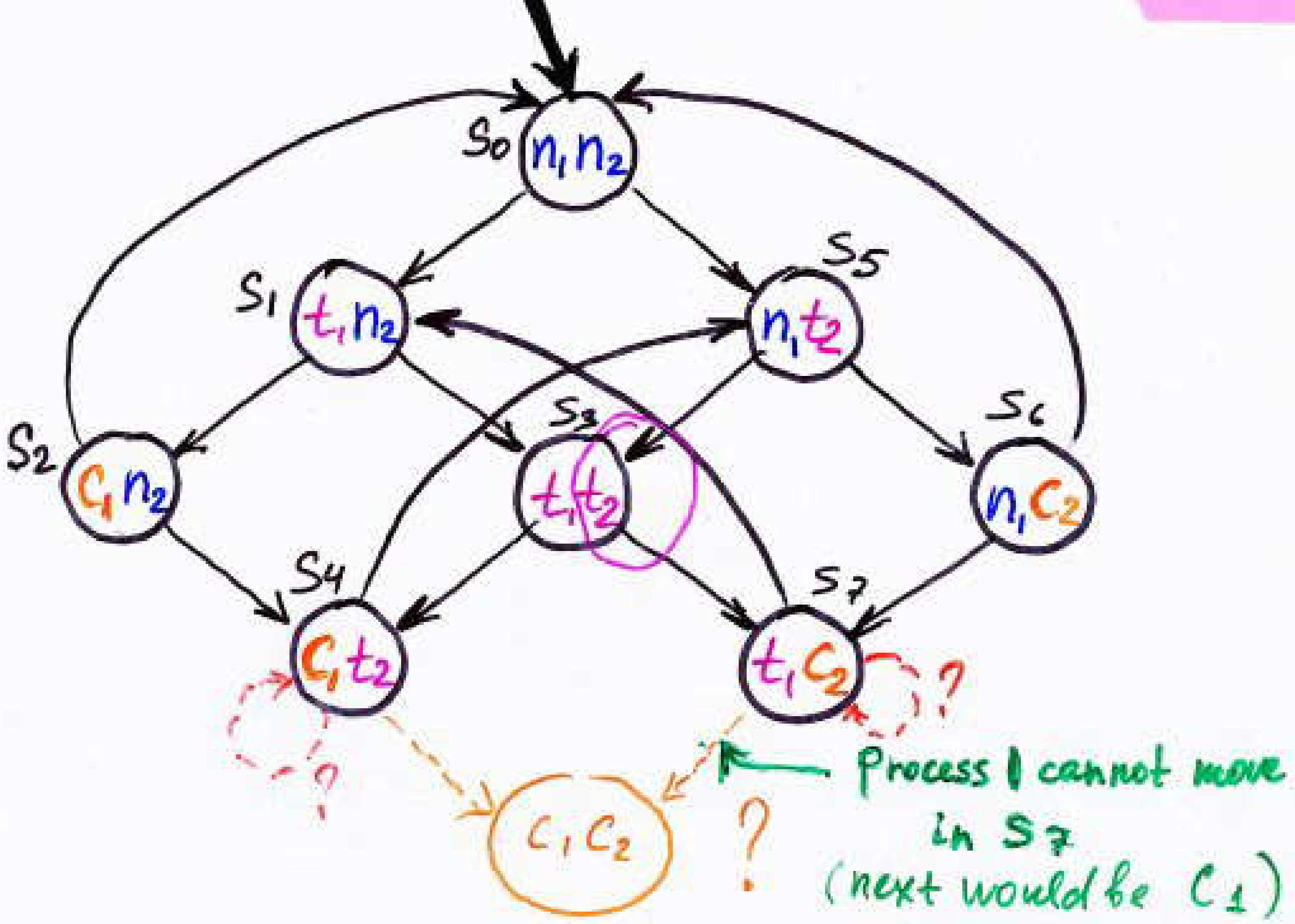
prc

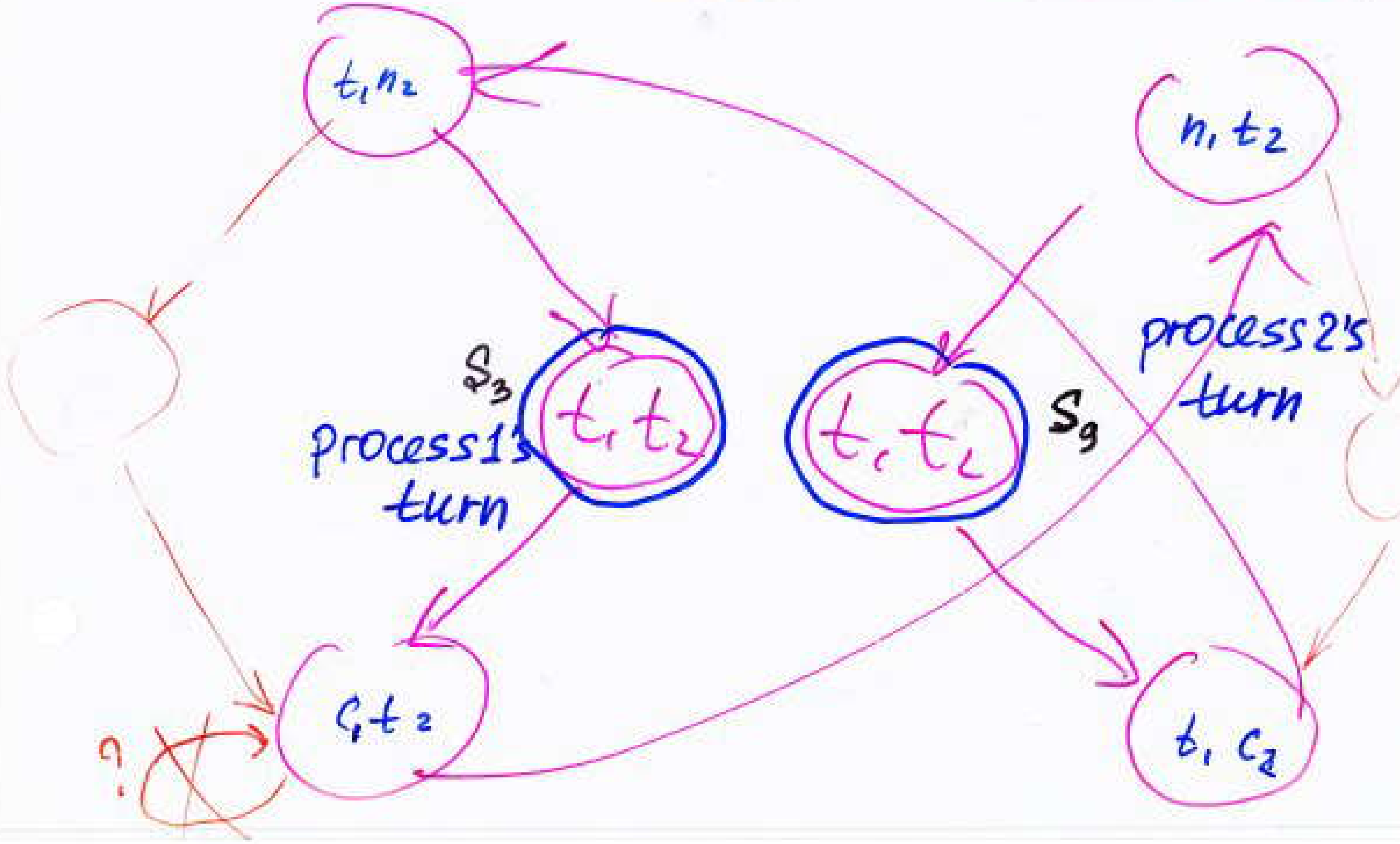
```
MODULE main
   VAR
      pr1: process prc(pr2.st, turn, 0);
      pr2: process prc(pr1.st, turn, 1);
      turn: boolean;
   ASSIGN
      init(turn) := 0;
   -- safety
   SPEC  AG!((pr1.st = c) & (pr2.st = c))
   -- liveness
   SPEC  AG((pr1.st = t) -> AF (pr1.st = c))
   SPEC  AG((pr2.st = t) -> AF (pr2.st = c))
   -- no strict sequencing
   SPEC EF(pr1.st=c & E[pr1.st=c U
                          (!pr1.st=c & E[! pr2.st=c U pr1.st=c ])])

MODULE prc(other-st, turn, myturn)
   VAR
      st: {n, t, c};
   ASSIGN
      init(st) := n;
      next(st) :=
         case
            (st = n)                                       : {t,n};
            (st = t) & (other-st = n)                      : c;
            (st = t) & (other-st = t) & (turn = myturn): c;
            (st = c)                                       : {c,n};
            1                                              : st;
         esac;
      next(turn) :=
         case
            turn = myturn & st = c : !turn;
            1                      : turn;
         esac;
   FAIRNESS running
   FAIRNESS  !(st = c)
```

Fig. 3.23. SMV code for mutual exclusion.

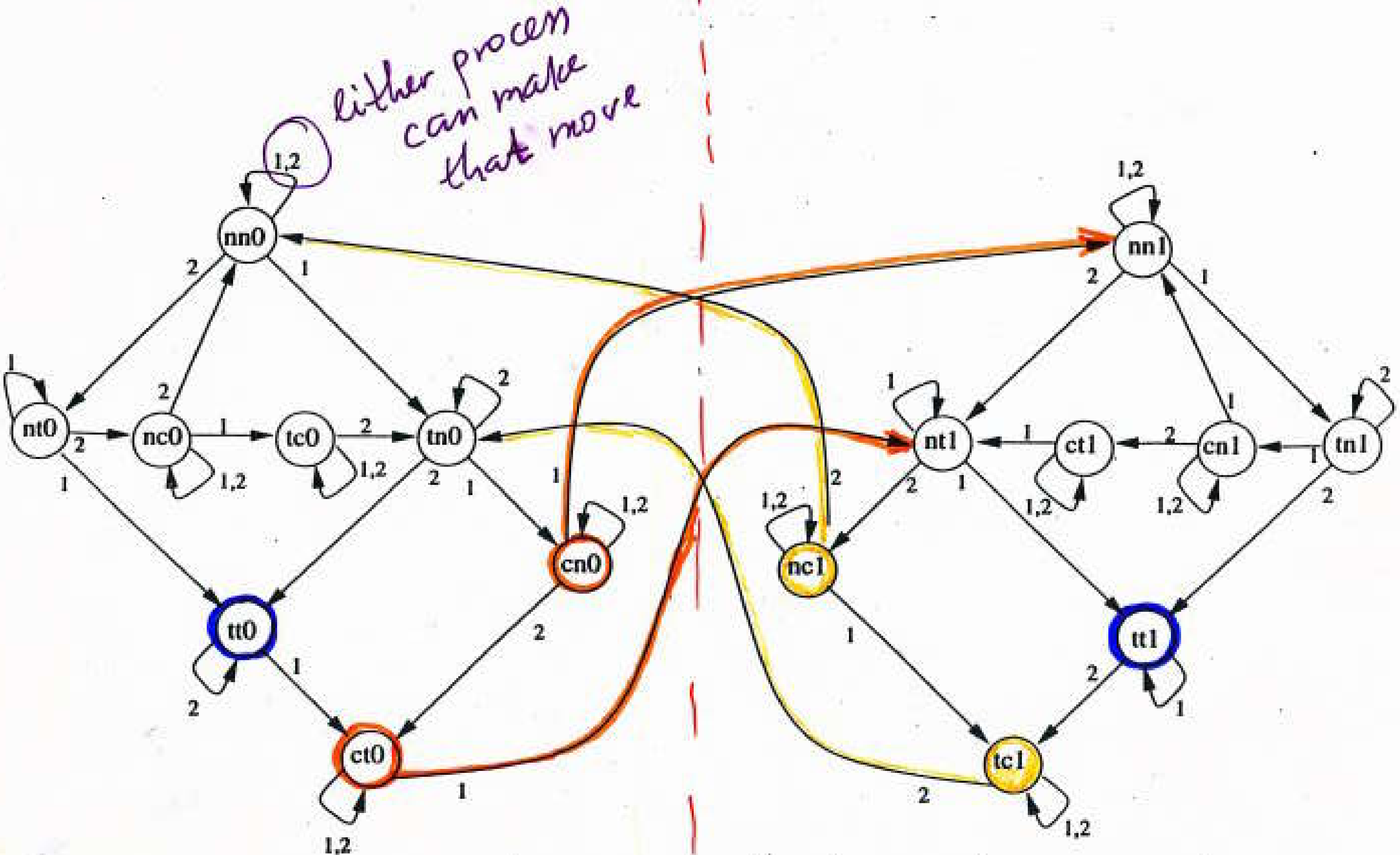
At each step, an uspecified scheduler determines which process may run





When $t_1 t_2$, it's the turn of Process 1 to enter c_1

When $t_1 t_2$, it's the turn of Process 2 to enter c_2



when in c, the turn changes

turn = 0

turn = 1

SMV: Synchronous and Asynchronous composition

By default, modules in SMV are composed $\boxed{\text{synchronously}}$;

∃ a global clock, and each time it ticks. each module is executed in parallel

process keyword:

allows to compose the modules $\boxed{\text{asynchronously}}$

run at different speed

at each tick, one module is non-deterministically chosen and executed for one cycle.

useful for:

communication protocols
asynchronous circuits
....

Model Checking with Fairness

Often, need to restrict the search tree to paths where some formula φ is true infinitely often.
(e.g. to model fair access to resourses)

If we have FAIRNESS φ

then, when checking spec. ψ, SMV ignores all states where φ does not hold inf. often.

Notice: liveness $AG(t_1 \Rightarrow AF c_1)$

will be violated if process stays in its critical section forever

add: FAIRNESS !st=c

(inf. often $st \neq c$)

restriction on the models not a SPEC!

As a result, the connectives A and E in the SPEC range over fair paths only!

This effect is not achived by adding

SPEC AG AF $\neg$(st=c)

SPEC is a question, not a restriction imposed on the model!

Note: Suppose we have

$$\boxed{\begin{array}{l}\text{FAIRNESS } !(st=c) \\ \text{SPEC } \psi\end{array}}$$

Q: Can we achive the same effect by asking (in CTL):

~~SPEC~~ AG

$$\text{SPEC}\quad \overbrace{\underbrace{AG\,AF\,!st=c}_{\text{false}} \rightarrow \underbrace{\psi}_{\text{false}}}^{\text{true}}$$

No! get answer "yes" even if ψ is false.

FAIRNESS running

restricts attention to execution paths along which the module in which it appears is selected for execution inf. often

usefull if want to ignore paths in which a module is starved of processor time

e.g. want to eliminate the case where an instance of prc (in Mutual Excl.) is never selected for execution (o.w. liveness would be violated)

$G(t \to Fc)$ $AG(t \to AFc)$

The Alternating Bit Protocol (ABP)

transmit messages along a "lossy line"
(may lose or duplicate messages)

If line does not lose inf. many messages, communication between the sender and the receiver will be successful

the line may not corrupt messages

4 agents:
- sender
- receiver
- message channel
- acknowledgement channel

Sender: transmits the first part of the message together with the control bit 0

Receiver sends 0 along the ack. channel

Sender: when receives this ackn., sends the next packet with 1

Both sender & receiver

ignore messages with unexpected bit

⇒ no duplicating, losing occur

Both sender & receiver continue the same action until the expected value arrives.

Sender

```
MODULE sender(ack)
VAR
    st         : {sending,sent};
    message1 : boolean;
    message2 : boolean;
ASSIGN
    init(st)  := sending;
    next(st)  := case
                    ack = message2 & !(st=sent) : sent;
                    1                           : sending;
                 esac;
    next(message1) :=
                 case
                    st = sent : {0,1};
                    1         : message1;
                 esac;
    next(message2) :=
                 case
                    st = sent : !message2;
                    1         : message2;
                 esac;
FAIRNESS running
SPEC AG AF st=sent
```

non-determ. choice (e.g. from the environment (e.g. user))

The sender must be selected to run inf. often

Receiver

```
MODULE receiver(message1,message2)
VAR
    st       : {receiving,received};
    ack      : boolean;
    expected : boolean;
ASSIGN
    init(st) := receiving;
    next(st) := case
                  message2=expected & !(st=received) : received;
                  1                                  : receiving
                esac;
    next(ack) :=
                case
                  st = received : message2;
                  1             : ack;
                esac;
    next(expected) :=
                case
                  st = received : !expected;   change truth value
                  1             : expected;    after receiving
                esac;
FAIRNESS running
SPEC AG AF st=received   can always succeed in receiving
  (Liveness)
```

Channels

1

```
MODULE one-bit-chan(input)
VAR
   output : boolean;
   forget : boolean;
ASSIGN
   next(output) := case
                        forget : output;
                        1:       input;
                   esac;
FAIRNESS running
FAIRNESS input & !forget
FAIRNESS !input & !forget
```

to model lossy character

the value of input gets transmitted to output, unless forget is true

2

```
MODULE two-bit-chan(input1,input2)
VAR
   forget : boolean;
   output1 : boolean;
   output2 : boolean;
ASSIGN
   next(output1) := case
                        forget : output1;
                        1:       input1;
                    esac;
   next(output2) := case
                        forget : output2;
                        1:       input2;
                    esac;
FAIRNESS running
FAIRNESS input1 & !forget
FAIRNESS !input1 & !forget
FAIRNESS input2 & !forget
FAIRNESS !input2 & !forget
```

to avoid dropping all 0 bits

although channels can lose messages, we assume that they inf. often transmit the message correctly (! forget)

Main

```
MODULE main
VAR
   S : process sender(ack_chan.output);
   R : process receiver(msg_chan.output1,msg_chan.output2);
   msg_chan : process two-bit-chan(S.message1,S.message2);
   ack_chan : process one-bit-chan(R.ack);
ASSIGN
   init(S.message2) := 0;
   init(R.expected) := 0;
   init(R.ack)           := 1;
   init(msg_chan.output2) := 1;
   init(ack_chan.output) := 1;

SPEC  AG(S.st=sent & S.message1=1 -> msg_chan.output1=1)
```

Receiver starts off by sending 1 as its ack. so that sender does not think that its first message is acknowledged

The first (main) message bit 1 has been sent then 1 has been received by the receiver

safety property