Introduction to SMV

Angelo gargantini
unibg

NuSMV: overview

- SMV = "Symbolic Model Verifier" (McMillan 1993).
- NuSMV is a reimplementation of SMV.
- NuSMV (http://nusvm.irst.itc.it) is an open source, automatic model checker.
- Joint effort IRST-ITC (Italy), CMU (USA), and others.
- It implements efficient techniques (ordered binary decision diagrams and SAT methods).
- It supports CTL (and LTL) model checking.

Symbolic Model Verifier

- Ken McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, 1993.
- Kripke structues described in a specialized language
- Specifications given as CTL formulas + Fairness
- Internal representation using ROBDDs
- Automatically verifies specification or produces a counterexample

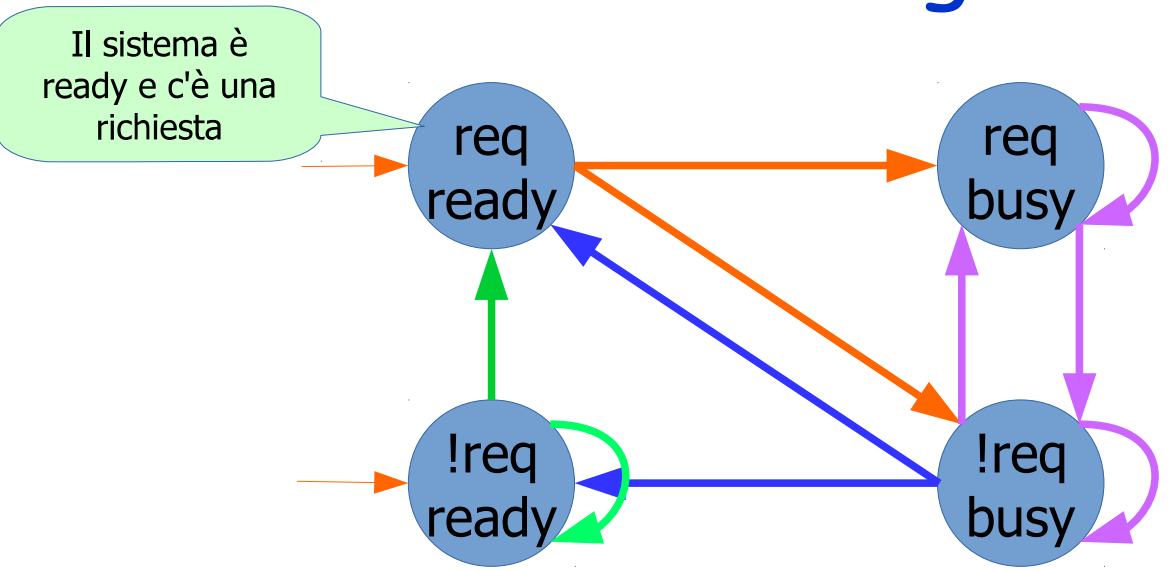
NuSMV basics

- Any model checker should provide at least:
  - An input language to describe the model Ms
  - And the properties φ to be checked
  - An automated mechanism for checking Ms|= φ
- The input language of NuSMV is called SMV

Language Characteristics

- Allows description of synchronous and asynchronous systems
- Modularized and hierarchical descriptions
- Finite data types: Boolean and enumerated
  - Integer only bounded (intervals)
- Nondeterminism

The model of Single Process



Un sistema che inizialmente è ready. Se arriva una richiesta va a busy. Dopo un po' torna ready (se non c'è una ulteriore richiesta)

Leggeremente diversa dal libro

A Sample SMV Program

```
MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
  case
     state=ready & request: busy;
     state=busy & request: busy;
     state=ready & not request: ready;
   TRUE: {ready, busy};
  esac;
SPEC AG(request -> AF (state = busy))
```

Declaration of main module

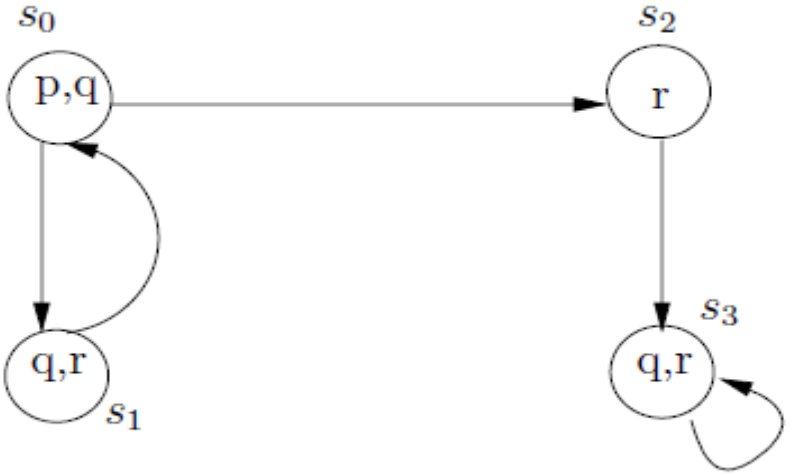
Declaration of variables

Initial state

State transition

formula to be checked

How to map a M into a NuSMV model




```
MODULE main
VAR
    p,q,r: boolean;
    state: {s0,s1,s2,s3};
ASSIGN
    init(state) := {s0,s1,s2,s3};
    next(state) :=
        case
            state=s0 & p & q & !r & !next(p) & !next(q) & next(r): s2;
            state=s0 & p & q & !r & !next(p) & next(q) & next(r): s1;
            state=s1 & !p & q & r & next(p) & next(q) & !next(r): s0;
            state=s2 & !p & !q & r & !next(p) & next(q) & next(r): s3;
            state=s3 & !p & q & r & !next(p) & next(q) & next(r): s3;
            1: state;
        esac;
```

NuSMV language

- One "main" module. Three portions of code, identified by VAR, ASSIGN, SPEC.
- VAR identifies a portion of code where variables are defined.
- ASSIGN identifies a portion of code where variables are initialized and evolution is described.
- SPEC defines properties to be verified.

Variable Assignments

- Assignment to initial state:
  init(value) := 0;
- Assignment to next state (transition relation)
  next(value) := value + carry_in mod 2;
- Assignment to current state (invariant)
  carry_out := value & carry_in;

- Use either init-next or invariant - never both

Types

- Boolean
  - TRUE is true and FALSE is false
- Enumerate
  - VAR
    a : {red, blue, green};
    b : {1, 2, 3};
    c : {1, 5, 7};
    ASSIGN
    next(b) := case
    b<3 : b+1;
    TRUE : 1;
    esac;
  - Numerical operations must be properly guarded otherwise they can go outside the domain
- Integers as intervals
  - A: 1 .. 100;

ASSIGN and DEFINE

- VAR a: boolean;
  ASSIGN a := b | c;
  - declares a new state variable a
  - becomes part of invariant relation

- DEFINE d:= b | c;
  - is effectively a macro definition, each occurrence of d is replaced by b | c
  - no extra BDD variable is generated for d
  - the BDD for b | c becomes part of each expression using d

Next

- Expressions can refer to the value of a variable in the *next* state
- Examples:
  - VAR a,b : boolean;
    ASSIGN
      next(b) := !b;
      a := next(b);

  - ASSIGN next(a) := !next(b)

    (a is the negation of b, except for the initial state)

    - Disclaimer: different SMV versions differ on this

Circular definitions

- … are not allowed!
- This is illegal:
  - a := next(b);
    next(b) := c;
    c := a;
- This is o.k.
  - init(a) := 0;
    next(a) := !b;

    init(b) := 1;
    next(b) := !a;

    init(c) := 0;
    next(c) := a & next(b);

The Case Expression

- case is an *expression*, not a statement

- Guards are evaluated sequentially.
- The first one that is true determines the resulting value
- If none of the guards are true, an *arbitrary* valid value is returned
  - Always use an *else* guard!

Nondeterminism

- Completely unassigned variable can model unconstrained input.
- {val_1, …, val_n} is an expression taking on any of the given values nondeterministically.
- Nondeterministic choice can be used to:
  - Model an implementation that has not been refined yet
  - Abstract behavior
  - Inputs of the system

Describing Automata

- A simplified elevator model
- Variable state definition:

```
MODULE main
VAR
  cabin: 0..3;
  dir: {up, down};
  request: array 0..3 of boolean;
```

- With no other variables, the system has 4x2x24 = 128 possible states

Describing Automata

- Describe the initial states:

```
ASSIGN
  init(cabin) := 0;
  init(dir) := up;
  init(request[0]) := 0;
    .....
```

Describing Automata

- Describe the transitions:

```
ASSIGN
next(cabin) := case
    dir=up & cabin<3:
                        cabin+1; -- moves up
    dir=down & cabin>0:
                        cabin-1; -- moves down
    TRUE : cabin;
 esac;
```

Describing Automata

- Describe the transitions:

```
next(dir) := case
    dir=up & next(cabin)=3:
                        down; -- switch dir
    dir=down & next(cabin)=0:
                        up; -- switch dir
    TRUE: dir;                  -- keep dir;
esac;
```

Per semplicità va sempre o up o down

Describing Automata

- A request for some floor appears at any time except if the cabin is actually on this floor, so the request cannot disappear until the cabin does reach that floor:

```
next(request[0]) := case
      next(cabin)=0 : FALSE;      -- disappears
      request[0] : TRUE;          -- remains
      TRUE : {FALSE,TRUE};        -- may appear;
                        esac;
next(request[1]) := case
      next(cabin)=1 : FALSE;      -- disappears
      request[1] : TRUE;          -- remains
      …
```

Devo ripetere la cosa per tutti gli elementi dell'array: posso farlo piu' rapidamente?

Describing Automata

- SMV offers another way to describe initial states and transitions in a declative way
- Initial states and transitins described by a chatacteristic property:

  INIT

  (dir = up & cabin =0);

  - for request[i] all values are possible

  TRANS

  ...

Verification properties

- SMV supports LTL formulas
- Verify that the elevator model has no deadlock:

  LTLSPEC

  G F dir = up ....

- Check that all requests are eventually satisfied:

  LTLSPEC

  ....

Verification properties

- SMV supports CTL formulas
- Verify that the elevator model has no deadlock:

```
SPEC
   AG EX TRUE
```

- Check that all requests are eventually satisfied:

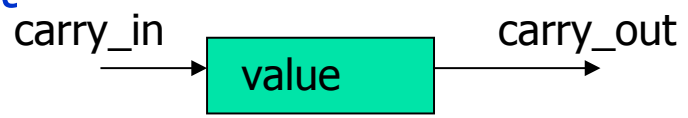
```
SPEC
      AG(AF!request[0] & AF!request[1] &
      AF!request[2] & AF!request[3])
```

Modules and Hierarchy

- Modules can be instantiated many times, each instantiation creates a copy of the local variables
- Each program has a module main
- Scoping
  - Variables declared outside a module can be passed as parameters
  - Internal variables of a module can be used in enclosing modules (submodel.varname).
- Parameters are passed by reference.

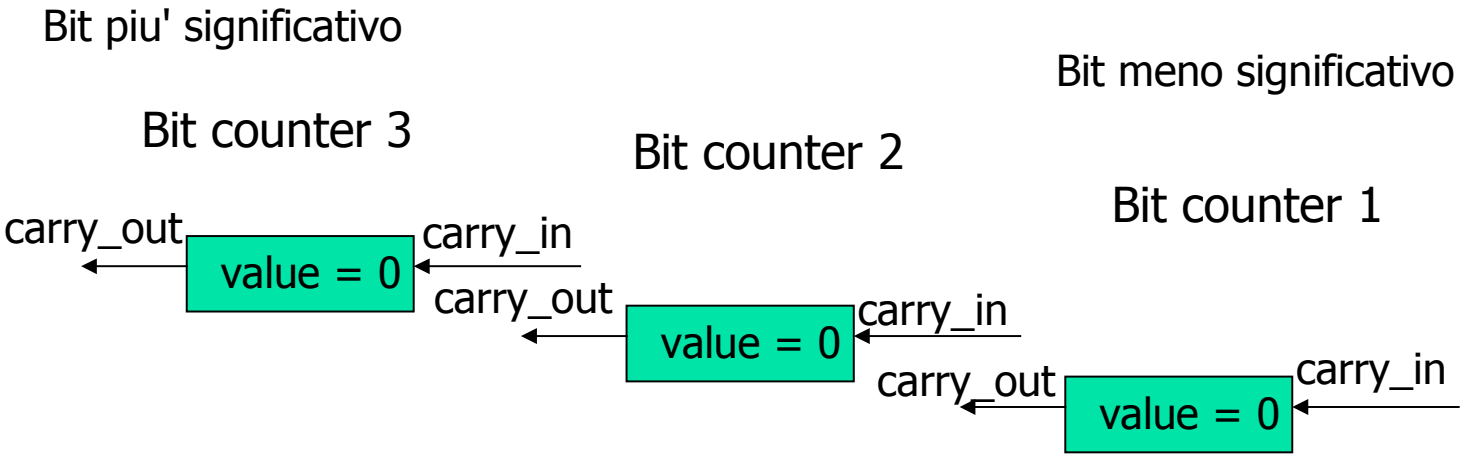
The bit counter example

- A counter repeatedly counts from 000 to 111 is described as three single-bit counters
- A single-bit counter add carry_in to value and reports in carry_out



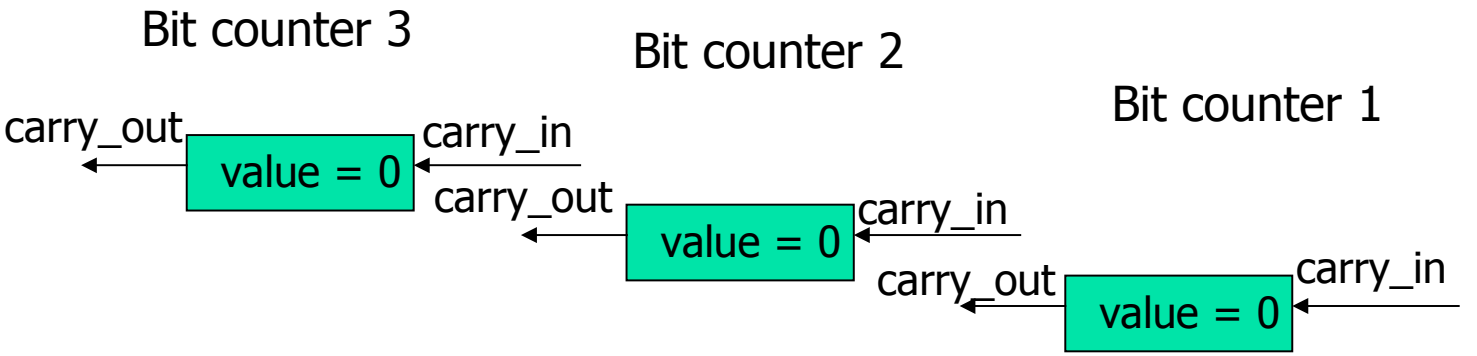
- If carry_in is TRUE, the counter adds 1 to its value (with a possible carry out to signal to the next counter)
  - Value is incremented if carry_in is TRUE
  - Carry_out signals that the next single bit counter must be incremented, iif value = TRUE and carry_in

The bit counter example



- parte da 000
- +1 in carry_in di bit counter 1
- passa a 001
- +1 in carry_in di bit counter 1
- passa a 010
- ecc

The bit counter example



The first counter has input 1, while the other two have as input the carry_out of the previous counter

The bit counter example

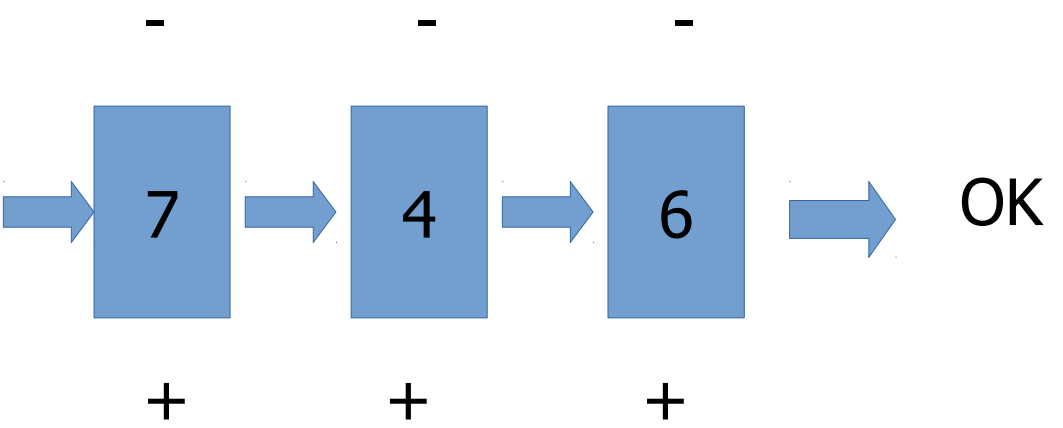
```
MODULE main
VAR
   bit0 : counter_cell(1);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
SPEC  ! F bit2.carry_out = 1

MODULE counter_cell(carry_in)
VAR   value : {0,1};
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE  carry_out := value * carry_in;
```

The bit counter example con boolean

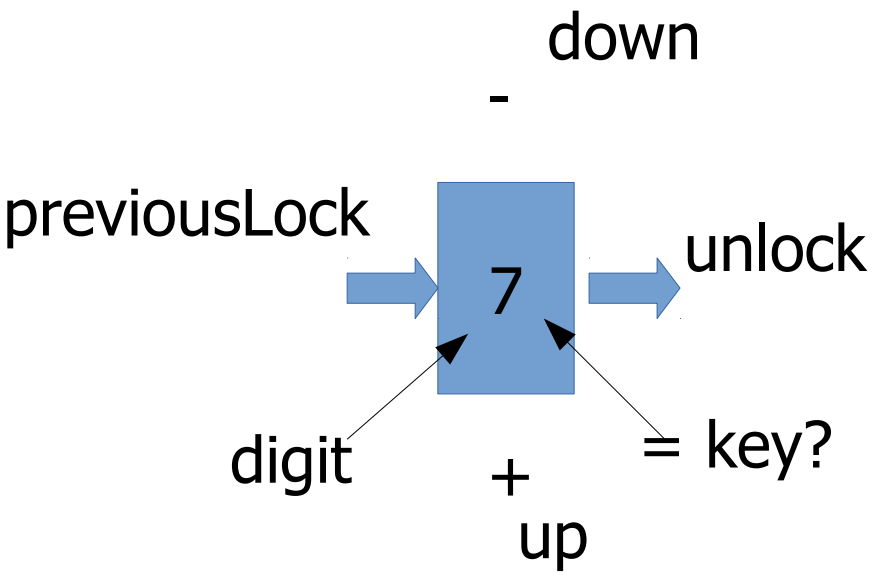
```
MODULE main
VAR
   bit0 : counter_cell(TRUE);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
SPEC  AG AF bit2.carry_out

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

The lock example (con moduli)

- Un lucchetto elettronico in cui ho una serie di cifre da indovinare.
  - Per comandare una cifra posso incrementare o dec il numero al display
  - Se indovino passo l'output al prossimo lucchetto ad una cifra



Single lock



none se non faccio nulla

Lock smv code

```
MODULE main
 VAR
   unlock1 : boolean;
   --unlock2 : boolean;
   lock1 : lock(7,TRUE,unlock1);
   lock2 : lock(2,unlock1,unlock2);

MODULE lock(key,previousLock,unlock)
VAR
digit : 0 .. 9;
command: {up,down,none};
ASSIGN
init(digit) := 0;
next(digit) := case
    command = none : digit;
    command = up: (digit +1) mod 10;
    command = down: (digit + 9) mod 10;
esac;
unlock := digit = key & previousLock;
```

Module Composition

- **Synchronous** composition
  - All assignments are executed in parallel and synchronously (a global clock).
  - A single step of the resulting model corresponds to a step in each of the components.

- **Asynchronous** composition
  - A step of the composition is a step by exactly one process
    - at each tick of clock one process is chosen non-deterministically and executed for one cycle.
  - Variables, not assigned in that process, are left unchanged.

Module Composition

- The bit-counter is synchronous
- Asynchronous composition is possible by means of the process keyword
- The following program represents a ring of three asynchronous inverting gates
- Among all the modules instantiated with the process keyword, one is nondeterministically chosen

Lock ansync

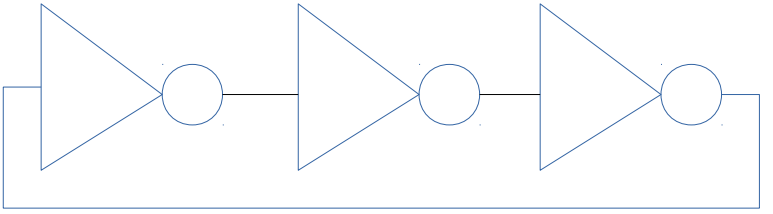
1. Every lock is a process:

```
lock1 : process lock(7,TRUE,unlock1);
...
```

2. It is superflous having a none command:

```
command: {up,down};
```

Inverter ring



- Three inverters that are placed in circle
- three asynchronous inverting gates

Asynchronous Composition

```
MODULE main
VAR
   gate1: process inverter(gate3.output);
   gate2: process inverter(gate1.output);
   gate3: process inverter(gate2.output);
SPEC  (AG AF gate1.output)
SPEC  (AG AF !gate1.output)

MODULE inverter(input)
VAR   output: boolean;
ASSIGN
    init(output) := 0;
    next(output) := !input;
```

Aletrnative: declarative specification

- The set of possible initial states is specified as a formula in the current state variables.
  - A state is initial if it satisfies the formula.
- The transition relation is directly specified as a propositional formula in terms of the *current* and *next* values of the state variables.
  - Any current state/next state pair is in the transition relation if and only if it satisfies the formula.
- These two functions are accomplished by the 'INIT' and 'TRANS' keywords

Asynchronous Composition

```
MODULE main
   VAR
        gate1 : inverter(gate3.output);
        gate2 : inverter(gate1.output);
        gate3 : inverter(gate2.output);
MODULE inverter(input)
   VAR
        output : boolean;
INIT
   output = 0
TRANS
   next(output) = !input | next(output) = output
```

Asynchronous Composition

- The system is not forced to choose a given process to execute, thus the output of a given gate may remain constant, regardless of its input.
- Therefore, the properties
  - SPEC (AG AF gate1.output)
  - SPEC (AG AF !gate1.output)

are both false

Counterexamples

-- specification AG AF (!gate1.output) is false
-- as demonstrated by the following execution
state 2.1:
gate1.output = 0 gate2.output = 0
gate3.output = 0

state 2.2:
[executing process gate1]

-- loop starts here --
state 2.3:
gate1.output = 1
[stuttering]

Fairness constraint

- In order to force a process to execute infinitely often, we can use a fairness constraint.
- A fairness constraint restricts the attention of the model checker to only those execution paths along which a given formula is true infinitely often.
- Each process has a special variable called running which is 1 if and only if that process is currently executing.

Fairness constraint

- By adding the declaration:

  FAIRNESS
  running

  we can effectively force every instance of inverter to execute infinitely often.
- The properties
  - SPEC (AG AF gate1.output)
  - SPEC (AG AF !gate1.output)

are both true

Fairness

- FAIRNESS ctl_formulae
  - Assumed to be true infinitely often
  - Model checker only explores paths satisfying fairness constraint
  - Each fairness constraint must be true infinitely often

Counter Revisited

If the counter has an input which can be both true or false, we must add fairness

```
MODULE main
VAR
   count_enable: boolean;
   bit0 : counter_cell(count_enable);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
SPEC  AG AF bit2.carry_out
FAIRNESS count_enable
```

Mutual exclusion

- Another example of asynchronous model.
- It uses a variable semaphore to implement mutual exclusion between two asynchronous processes.
- 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 0, it goes to the critical state, and sets semaphore to 1.
- On exiting its critical region, the process sets semaphore to 0 again.

Mutual exclusion

```
MODULE main
  VAR
      semaphore : boolean;
      proc1 : process user(semaphore);
      proc2 : process user(semaphore);
  ASSIGN
      init(semaphore) := 0;
MODULE user(semaphore)
  VAR
      state : {idle, entering, critical, exiting};
  ASSIGN
      init(state) := idle;
```

Mutual exclusion

```
  next(state) := case
       state = idle : {idle, entering};
       state = entering & !semaphore : critical;
       state = critical : {critical, exiting};
       state = exiting : idle;
       1 : state;
       esac;
  next(semaphore) := case
       state = entering : 1;
       state = exiting : 0;
       1 : semaphore;
       esac;
FAIRNESS
  running
```

Mutual exclusion Properties

We require:

- Safety: only one process is in his critical section at any time
- Liveness : whenever any process requests to enter its critical section, it will eventually be permitted to do so

Mutual exclusion Properties

We require:

- Safety:

AG ! (proc1.state = critical & proc2.state = critical)

- Liveness :

AG (proc1.state = entering -> EF proc1.state = critical)

AG (proc2.state = entering -> EF proc2.state = critical)

Mutual exclusion Properties

- Liveness :

AG (proc1.state = entering -> EF proc1.state = critical)

AG (proc2.state = entering -> EF proc2.state = critical)

Note that:

AG (proc1.state = entering -> **AF** proc1.state = critical)

only if FAIRNESS proc1.state = critical

The ferryman problem

- A ferryman, a goat, a cabbage and a wolf are on one side of a river.
- The ferryman can cross the river with at most one passinger in the boat.
- There is a behaviour conflict between:
  - The goat and the cabbage
  - The goat and the wolf
- Can the ferryman transport all goods to the other side without any conflict?

The ferryman problem

- Four agents:
  - ferryman, goat, wolf, cabbage
- The location of each variable is modelled by a boolean value:
  - F denotes the agant is on the initial side
  - T denotes the agant is on the final side
- The variable carry indicates which good is carried by the ferryman

The ferryman problem

```
MODULE main
VAR
   ferryman : boolean;
   goat : boolean;
   cabbage : boolean;
   wolf : boolean;
   carry : {g,c,w,0};
ASSIGN
   init(ferryman) := 0;
   init(goat) := 0;
   init(cabbage) := 0;
   init(wolf) := 0;
   init(carry) := 0;
```

The ferryman problem

next(ferryman) := {0,1};

the ferryman can decide to cross the river or not

the value of carry is non deterministic but determined by the value of ferryman, goat, wolf, cabbage

The ferryman problem

```
next(carry) := case
ferryman=goat : g; g is a member of set from
1 : 0;          which next(carry) is chosen
esac union
case ferryman=cabbage : c;
1 : 0;
esac union
case ferryman=wolf : w;
1 : 0;
esac union 0;
```

the next value of goat, cabbage, wolf are deterministic, since whether they are carried or not is determined by the ferryman'choice represented by carry

The ferryman problem

```
next(goat) := case
   ferryman=goat & next(carry)=g : next(ferryman);
   1 : goat;
   esac;
next(cabbage) := case
   ferryman=cabbage & next(carry)=c : next(ferryman);
   1 : cabbage;
   esac;
next(wolf) := case
   ferryman=wolf & next(carry)=w : next(ferryman);
   1 : wolf;
   esac;
```

The ferryman problem

We like to find a path satisfying:

- the final reachability condition:

  cabbage & goat & wolf & ferryman

- under the safety condition:

  goat=cabbage | goat=wolf) -> goat=ferryman

SPEC

E (((goat=cabbage | goat=wolf) -> goat=ferryman)

U (cabbage & goat & wolf & ferryman))

This is true but why???

The ferryman problem

We run SMV with the **negation** of the property hoping to find a counter example:

```
! (E((goat=cabbage | goat=wolf) ->
   goat=ferryman)
        U (cabbage & goat & wolf & ferryman)))
```

However, SMV finds the following loop:

The ferryman problem

-> State 1.1 <-

ferryman = 0

goat = 0

cabbage = 0

wolf = 0

carry = 0

-> State 1.2 <-

ferryman = 1

goat = 1

carry = g

The ferryman problem

-> State 1.3 <-

ferryman = 0

carry = 0

-> State 1.4 <-

ferryman = 1

cabbage = 1

carry = c

The ferryman problem

-> State 1.5 <-

ferryman = 0

goat = 0

carry = g

-> State 1.6 <-

ferryman = 1

wolf = 1

carry = w

The ferryman problem

-> State 1.7 <-

ferryman = 0

carry = 0

-> State 1.8 <- (satisfy the property!!!)

ferryman = 1

goat = 1

carry = g

The ferryman problem

-> State 1.9 <-

-> State 1.10 <-

ferryman = 0

cabbage = 0

carry = c

-> State 1.11 <-

ferryman = 1

carry = 0

The ferryman problem

-> State 1.2 <-
ferryman = 0
wolf = 0
carry = w
-> State 1.13 <-
ferryman = 1
carry = 0
-> State 1.14 <-
ferryman = 0
goat = 0
carry = g
-> State 1.15 <-
carry = 0

The ferryman problem

SMV finds an infinite path which loops around the 15 states.

Along the infinite path, the farryman repeatedly takes his goods across (safely), and then back again (unsafely).

The property asserts the safety of the forward journey but say nothing about what happens after that.

The ferryman problem

The correct property is

SPEC

E (((goat=cabbage | goat=wolf) -> goat=ferryman)

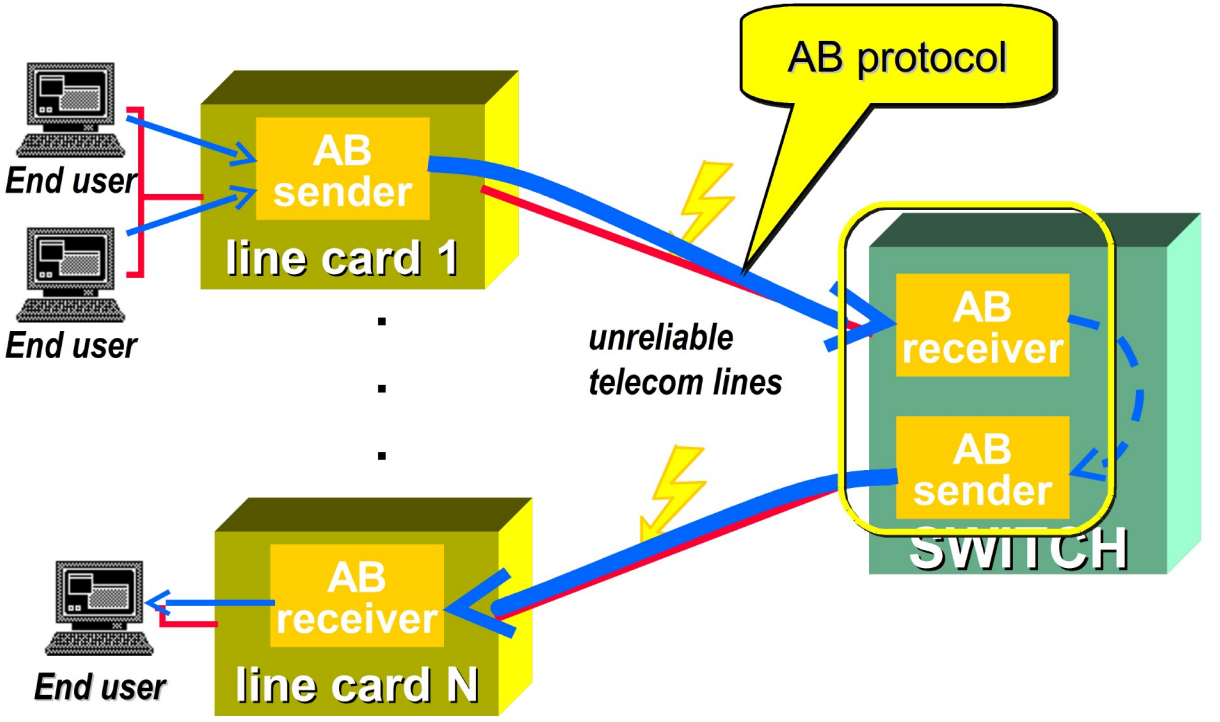
U (cabbage & goat & wolf & ferryman)

& AG(goat -> AG goat))

the goat makes at least three trips, and once it has crossed, it remains across

Case Study: Protocol Handler

- A multi-line packet switch that uses the alternating-bit protocol as its link protocol



The alternating bit protocol

- ABP is a protocol for transmitting messages along a "loose line"(a line which may lose or duplicate messages)
- Proving the line doesn't lose infinitely many messages, the protocol guarantees that communication between sender and reveiver will be successful.
  - we allow losing or duplication of messages but not corruption

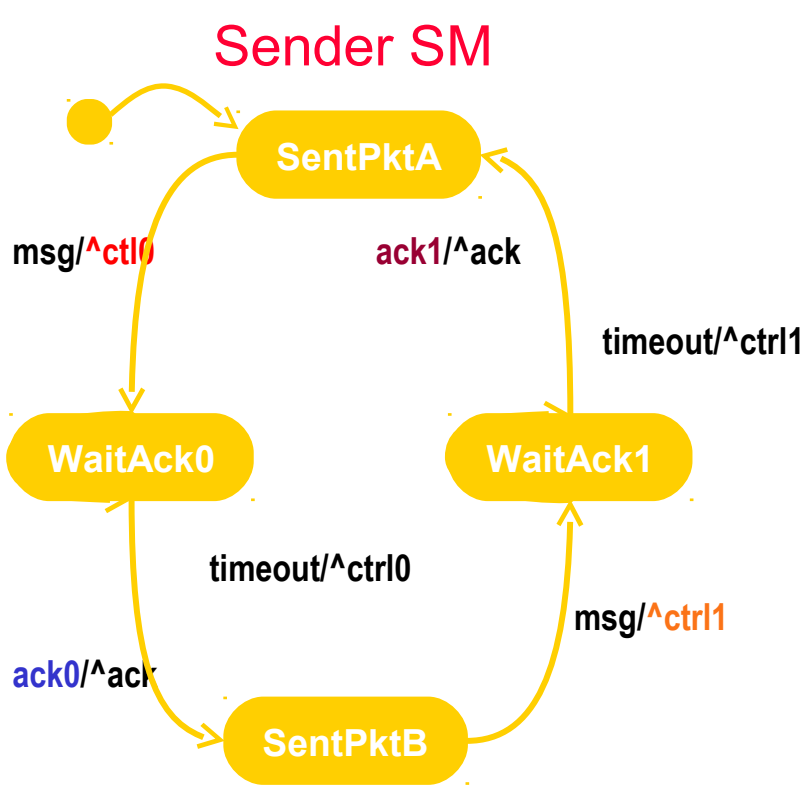
The alternating bit protocol

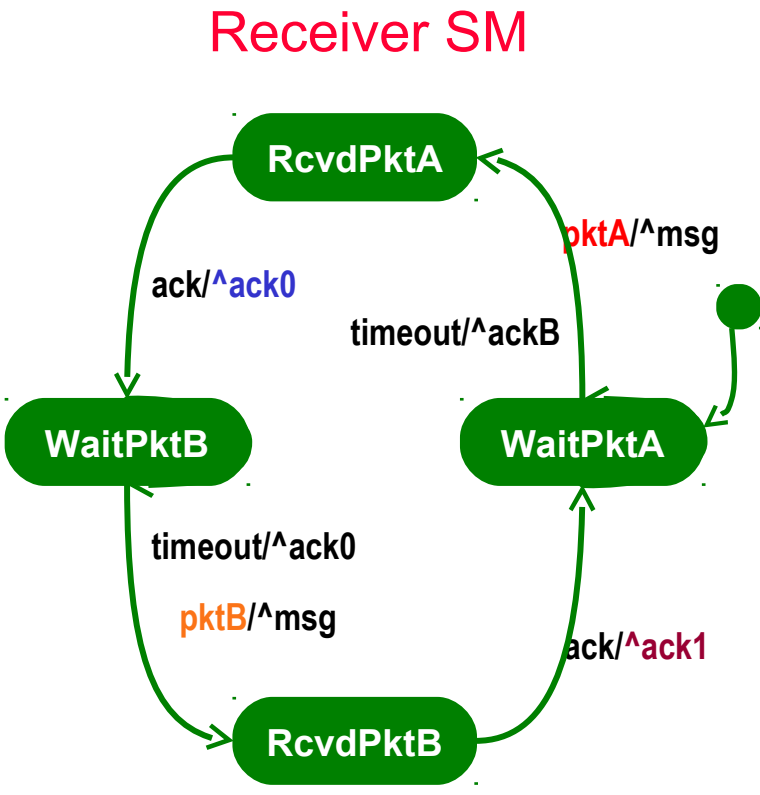
- There are 4 agents:
  - the sender
  - the receiver
  - the message channel
  - the acknowledgement channel

The alternating bit protocol

- The **sender** sends the first part of the packet + the ctrl-bit 0 along the msg-channel
- If and when the **receiver** gets a message with ctrl-bit 0, it sends 0 along the ack-channel
- When the **sender** receives the ack, it sends the next packet + ctrl-bit 1
- If and when the **receiver** gets the a msg with ctrl-bit 1, it sends 1 along the ack-channel
- If the **sender** doesn't receive the expected ack, it continually resends the message
- If the **receiver** doesn't get the message with the expected ctrl-bit, it continually resends the previous ack

Alternating Bit Protocol





The alternating bit protocol: sender

```
MODULE sender(ack)
VAR
   st : {sending,sent};
   message1 : boolean;
   message2 : boolean;
```

- We assume the packet to be sent is divided up into single-bit message which are sent sequentially
- message1 is the current bit of the msg
- message2 is the control bit
- The sender goes in mode sent only when recives the ack of the msg it has been sending

The alternating bit protocol: sender

```
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;
```

The alternating bit protocol:sender

FAIRNESS running

SPEC
   AG AF st=sent -- liveness

The alternating bit protocol: receiver

```
MODULE receiver(message1,message2)
VAR
   st : {receiving,received};
   ack : boolean;
   expected : boolean;
```

The alternating bit protocol : receiver

```
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;
   1 : expected;
   esac;
```

The alternating bit protocol : receiver

FAIRNESS running

SPEC

AG AF st=received -- liveness

The alternating bit protocol: one-bit-channel

```
MODULE one-bit-chan(input)
VAR
   output : boolean;
   forget : boolean; -- specifies lossy char

ASSIGN
   next(output) := case
   forget : output; -- input is transmitted to output unless forget is true
   1: input;
   esac;
FAIRNESS running
FAIRNESS input & !forget
FAIRNESS !input & !forget
```

- **ack-channel** is an instance of the one-bit-channel

The alternating bit protocol : two-bit-channel

```
MODULE two-bit-chan(input1,input2)
VAR
   forget : boolean; -- specifies lossy char
   output1 : boolean;
   output2 : boolean;
ASSIGN
   next(output1) := case
        forget : output1;
        1: input1;
        esac;
   next(output2) := case
        forget : output2;
        1: input2;
        esac;
```

The alternating bit protocol : two-bit-channel

FAIRNESS running
FAIRNESS input1 & !forget
FAIRNESS !input1 & !forget
FAIRNESS input2 & !forget
FAIRNESS !input2 & !forget

- Fairness constrants are intended to model the fact that, although channels can lose messages, they infinitively often transmit the message correctly
  - Without this constraint liveness can not be proved
- Fairness constrant "infinitively often !forget", is not sufficient since it doesn't prevent to drop all 0-bits and send only all 1-bits.

The alternating bit protocol : 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;
```

The alternating bit protocol : properties

- Safety: if the msg bit 1 has been sent and the correct ack has been returned, then a 1 was received by the rceiver

  SPEC

  AG (s.st=sent & s.message1=1 ->
  msg_chan.output1=1)

- Liveness:
  - messages get througt eventually

  SPEC AG AF s.st=sent

  - acknowledgenets get througt eventually

    SPEC AG AF r.st=received