Lecture 3

SPIN and Promela

What is SPIN(Simple Promela Interpreter)

A tool for analyzing models of reactive systems

Models described in Promela

- Language with concurrent processes,
- Communication via channels,

Analysis by

- Simulation
- Model checking
- Several optimizations implemented
- ”most efficient tool for explicit-state model checking”

Material About SPIN

SPIN Home page `http://spinroot.com/spin/`

Contains manuals, reference material, and tutorial

Books about SPIN

- G.J. Holzmann *SPIN MODEL CHECKER Primer and Reference Manual*,
- G.J. Holzmann *Design and Validation of Computer Protocols*,, Prentice Hall 1991, older book, available on the Internet

Elements of Promela

- Language for defining finite-state transition systems
- Data types with precisely defined finite domains
  - Bits, integers, arrays, messages
- Processes, which can be dynamically created
- Communication via global variables or communication channels
- Simple control constructs

Typical Structure of Promela Model

Promela model consists of

- type declarations
- channel declarations
- variable declarations
- process declarations
- **`init`** process

```
mtype = {msg, ack};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender () {
    ...
}
proctype Receiver () {
    ...
}

init {
       ...
}
```

Basic Variables and Types

Basic types

```
bit                    [0 .. 1]
bool                   [0 .. 1]
byte                 [0 .. 255]
short          [-2^15 .. 2^15 -1]
int            [-2^31 .. 2^31 -1]
```

Array declaration
Array access

```
byte anarray[24];
anarray[v] = anarray[3];
```

Records type definition

```
typedef Msg{
        byte a[3], b;
        chan p
}
```

Record declaration
Record access

```
Msg astruct
astruct.a[1]
```

Enumeration type f. messages

```
mtype = {ack, nak, err, next}
```

Expressions

Operators

```
+       -       *       /       %       ^
>       >=      <       <=      ==      !=
!       &&      ||
&       |       -
>>      <<      ++      --
```

Conditional expression

```
(v >= 0 ->  v : -v)
```

Operations on channel identifers

```
len(qid)
empty(qid)    nempty(qid)
full(qid)     nfull(qid)
```

Basic Statements

Expressions	`(y == false || x > 9)`
Assignments	`y = 34 % 3` `anarray[0] = anarray[3] * anarray[(v+2)/4]`
No-Op	`skip`
Goto	`goto label`
Print (only in simulation)	`printf`

- All basic statements are either executable (enabled) or blocked (disabled) (of course, depending on values of variables, etc.)
- Expressions are also statements
  - Blocked if evaluated to 0, otherwise executable
  - In this way, expressions/statements can be used as guards

Compound Statements

Sequential composition

Selection

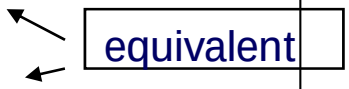
Selection can be nonderministic

```
test == 1 ; state = state + 1
test == 1 -> state = state + 1

if
:: (a == b) -> state = state + 1
:: (a != b) -> state = state - 1
fi

if
:: (a == b) -> state = state + 1
:: else     -> state = state - 1
fi

if
:: input ? offhook
:: (a == b) -> b = 3 ; goto onhook
:: output ! wakeup
:: b = 3  /*any statement can be guard*/
fi
```



Compound Statements (ctd.)

Repetition

```
do
:: (m > n)  -> m = m - n
:: (m < n)  -> n = n - m
:: (m == n) -> break
od ;
printf ("GCD = %d\n", m)
```

Processes and process types

Name

Local variable declarations

```
proctype Sender(chan in; chan out) {
    bit sndB, rcvB;
    do
    :: out ! data(sndB) ->
           in ? ack(rcvB);
           if
           :: sndB == rcvB -> sndB = 1-sndB
           :: else -> skip
           fi
    od
}
```

formal parameters

Processes defined by **`proctype`** definitions

A process type may be instantiated several times

Each process has its local state (pc, local variables)

Processes execute concurrently, by interleaving

Process instantiations

Processes are created by **`run`** statement/expression

`run` returns process id

Processes execute their first statement some time after creation

Processes can be statically created at initialization time (no formal parameters allowed)

```
proctype Foo(byte x) {
    ...
}

init {
    int pid = run Foo(2);
    run Foo(27)
}

active[3] proctype Bar(){
     ...
}
```

Communication Channels

Channels used for passing messages

- Asynchronous (buffered, default is FIFO)
- Synchronously (rendez-vous)

```
chan qid       = [4] of {mtype, int, byte}
chan synch[3]  = [0] of {mtype, int}
```

name capacity types of messages

Sending

Receiving

Non-modifying receive

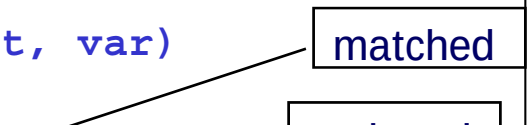
Sorted send/receive
(inserts lexicographically,
receives any element)

```
qid ! var1, const, var
qid ! err(const, var)

qid ? err(const, var)

qid ? [err , const, var]

qid !! err(const, var)
qid ?? err(const, var)
```



Communication Channels (ctd.)

Declaring synchronous (rendez-vous) channel w. capacity 0

```
chan synch[3] = [0] of {mtype, int}
```

Peterson-Fischer Mutual Exclusion

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l1: y1 = true;
l2: t = turn2;
l3: (y2 == false ||  t == turn1) ;
l4:  /* critical section */
    atomic{ y1 = false ; goto l1 }
}
```

```
proctype P2() {
m1: y2 = true;
m2: t = turn1;
m3: (y1 == false ||  t == turn2) ;
m4:  /* critical section */
    atomic{ y2 = false ; goto m1 }
}

init {
  atomic { run P1() ; run P2() }
}
```

Peterson-Fischer Mutual Exclusion

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l1: y1 = true;
l2: t = turn2;
l3: (y2 == false ||  t == turn1) ;
    mutex++ ;
    assert (mutex <= 1) ;
l4:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l1 }
}
```

```
proctype P2() {
m1: y2 = true;
m2: t = turn1;
m3: (y1 == false ||  t == turn2) ;
    mutex++ ;
    assert (mutex <= 1) ;
m4:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m1 }
}

init {
  atomic { run P1() ; run P2() }
}
```

Problem with Atomicity

Only basic statements are atomic

Q: What is the final value of **`state`** ?

```
byte state = 1;

proctype A() {
    state == 1 -> state++
}

proctype B() {
    state == 1 -> state--
}

init {
    run A() ; run B()
}
```

Solution: atomic-construct

A process whose control is inside

`atomic{ }`

executes without being interrupted by other processes

NOTE: Make sure that such a sequence cannot be blocked inside (after the first statement).

In that case, Promela will suspend the process, and you get unintended semantics.

```
byte state = 1;

proctype A() {
    atomic {
        state == 1 -> state++
    }
}

proctype B() {
    atomic {
        state == 1 -> state--
    }
}

init {
    atomic {run A() ; run B()}
}
```

Other use of `atomic{ }`

To group complex manipulation into single transition

If the manipulation is deterministic and always exits at the end

`d_step {`

is more efficient

```
atomic {
        cnt = 0 ;
        do
        :: (cnt < Max) -> z[cnt] = 3; cnt++
        :: (cnt >= Max) -> break
        od
}

d_step {
        cnt = 0 ;
        do
        :: (cnt < Max) -> z[cnt] = 3; cnt++
        :: (cnt >= Max) -> break
        od
}
```

Else and Timeout

`else` is enabled if no other statement in the same process is enabled

`timeout` is enabled if no other statement in the entire Promela model is enabled

`skip` is best to use if we want to be sure to analyze the effect of possibly premature timeout.

```
do
:: (m > n)  -> m = m - n
:: (m < n)  -> n = n – m
:: else -> break
od ;
printf (”GCD = %d\n”, m)

do
:: input ? offhook
:: input ? ringing
:: timeout -> output ! wakeup
Od

do
:: input ? offhook
:: input ? ringing
:: skip -> output ! wakeup
od
```

Useful Macros

IF without else branch

```
#define IF   if ::
#define FI   :: else fi
```

Allows to write

```
IF b -> x++ FI
```

FOR loop

```
#define FOR(i,l,h)  i = l ; do :: i < h ->
#define ROF(i,l,h)  ;i++ :: i >= h -> break od
```

Allows to write

```
FOR(i,0,N) run proc(i) ROF(i,0,N)
```

which means

```
i = 0 ;
do
:: i < N  -> run proc(i)
:: i >= N -> break
od
```

Message Transmission Protocol

```
mtype = { m0, m1, ack}



proctype Sender{
Send0:  do
        :: skip ->   /* timeout */
                StoR!m0
        :: RtoS?ack ->  /* rec ack */
                SoR!m1;
                goto Send1
        od;
Send1:  do
        :: skip ->    /* timeout */
                StoR!m1
        :: RtoS?ack ->  /* rec ack */
                StoR!m0;
                goto Send0
        od
}
```

```
proctype Receiver{
Rec0:   do
        :: StoR?any -> skip  /* loss */
        :: StoR?m0  -> RtoS!ack;
                       goto Rec1
        od
Rec1:   do
        :: StoR?any -> skip  /* loss */
        :: StoR?m1  -> RtoS!ack;
                       goto Rec0
        od
}

init {
    chan StoR = [1] of { mtype };
    chan RtoS = [1] of { mtype };
    atomic {
            StoR!m0;  /* start */
            run Sender;
            run Receiver
    }
}
```

Alternating Bit Protocol (version 1)

```
mtype    = { msg, ack };

chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit) ->
           r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                 sbit = 1 - sbit;
                 data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd ;
    bit rbit, seqno = 0;
    do
    :: s_r ? msg (recd, seqno) ->
           r_s ! ack(seqno);
           if
           :: seqno == rbit ->
                 rbit = 1 - rbit
           :: else
           fi
    od
}
```

Alternating Bit Protocol (version 1 altern)

```
mtype    = { msg, ack };

chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit) ->
           r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                 sbit = 1 - sbit;
                 data++
           :: else
           fi
    od
}
```

```
proctype Receiver() {
    byte recd ;
    bit rbit, seqno = 0;
    do
    :: s_r ? msg (recd, seqno) ->
           r_s ! ack(seqno);
           if
           :: seqno == rbit ->
                 rbit = 1 - rbit
           :: else
           fi
    od
}

init {
    atomic {
           run Sender();
           run Receiver()
    }
}
```

AB Protocol (version 2, with losses)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit) ->
           r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    :: (1) -> skip
    od
}
```

```
active proctype Receiver() {
    byte recd ;
    bit rbit, seqno = 0;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: r_s ! ack(seqno)
           :: skip
           fi;
           if
           :: seqno == rbit ->
                  rbit = 1 - rbit
           :: else
           fi
    od
}
```

AB Protocol (version 3, w. retransmission)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: s_r ! msg(data, sbit)
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
                  assert(recd ==
    expected) ;
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> skip
    od
}
```

AB Protocol (version 4, w. checks)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: data < 10 -> s_r ! msg(data,
    sbit)
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
                  assert(recd == expected);
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> skip
    od
}
```

AB Protocol (version 5, limiting checks)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: data < 10 -> s_r ! msg(data,
    sbit)
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
  progress:   assert(recd == expected) ;
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> skip
    od
}
```

AB Protocol (version 6, w progress)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: data < 10 -> s_r ! msg(data,
    sbit)
    :: (1) -> progress1: skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
  progress:   assert(recd == expected) ;
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> progress2: skip
    od
}
```

AB Protocol (version 7, w progress)

```
# define MAXMSG 4

mtype       = { msg, ack };
chan s_r    = [2] of {mtype , byte,
   bit};
chan r_s    = [2] of {mtype , bit };
chan source = [0] of {byte};
chan sink   = [0] of {byte};

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    source?data;
    do
    :: s_r ! msg(data, sbit)
    :: (1) -> progress1: skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  source?data
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                sink!recd;
                progress: rbit = 1 - rbit
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> progress2: skip
    od
}
```

AB Protocol (version 7, w harness)

```
# define MAXMSG 4

chan source = [0] of {byte};
chan sink   = [0] of {byte};




active proctype Generator() {
    byte seed = 0;
    do
    :: source!seed -> seed =
    (seed+1)%MAXMSG
    od
}
```

```
active proctype Checker() {
    byte expected, inmsg= 0;
    do
    :: sink?inmsg ->
           assert(inmsg == expected);
           expected = (expected+1)%MAXMSG
    od
}
```

AB Protocol (version 8, new harness)

```
# define white  0
# define red    1
# define blue   2

chan source = [0] of {byte};
chan sink   = [0] of {byte};


active proctype Generator() {
    do
    :: source!white
    :: source!red -> break
    od;
    do
    :: source!white
    :: source!blue -> break
    od;
end: do
    :: source!white
    od
}
```

```
active proctype Checker() {
    byte inmsg;
    do
    :: sink?inmsg ->
        if
        :: (inmsg == red) -> break
        :: else assert(inmsg == white)
        fi
    od;
    do
    :: sink?inmsg ->
        if
        :: (inmsg == blue) -> break
        :: else assert(inmsg == white)
        fi
    od;
end1: do
    :: sink?inmsg ->
        assert(inmsg == white)
    od
}
```

AB Protocol (version 9, w acceptance)

```
#define p Sender@Slabel
#define q Receiver@Rlabel
#define r Receiver@Rsuccess

mtype       = { msg, ack };
chan s_r    = [2] of {mtype , byte,
    bit};
chan r_s    = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    source?data;
    do
    :: s_r ! msg(data, sbit) ->
    Slabel: skip
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  source?data
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                Rsuccess: sink!recd;
                rbit = 1 - rbit
           :: else
           fi
    :: r_s ! ack (rbit) -> Rlabel: skip
    :: (1) -> skip
    od
}
```

AB Protocol (version 9, the never claim)

```
#define p Sender@Slabel
#define q Receiver@Rlabel
#define r Receiver@Rsuccess

/*
     * Formula As Typed: ([]  <>  p &&
    []  <>  q) -> []  <>  r
     * The Never Claim Below
    Corresponds
     * To The Negated Formula !(([]  <>
    p && []  <>  q) -> []  <>  r)
     * (formalizing violations of the
    original)
     */

never {    /* !(([]  <>  p && []  <>
    q) -> []  <>  r) */
```

```
T0_init:
    if
    :: (! ((r)) && (p) && (q)) -> goto
    accept_S485
    :: (! ((r)) && (p)) -> goto T2_S485
    :: (! ((r))) -> goto T0_S485
    :: (1) -> goto T0_init
    fi;
accept_S485:
    if
    :: (! ((r))) -> goto T0_S485
    fi;
T2_S485:
    if
    :: (! ((r)) && (q)) -> goto
    accept_S485
    :: (! ((r))) -> goto T2_S485
    fi;
T0_S485:
    if
    :: (! ((r)) && (p) && (q)) -> goto
    accept_S485
    :: (! ((r)) && (p)) -> goto T2_S485
    :: (! ((r))) -> goto T0_S485
    fi;
}
```

Specifying invariant properties

Always executable

If `v <= 2` is false,

SPIN exits with error

Used to check invariants

```
assert (v <= 2)
```

Checking by monitor

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l1: y1 = true;
l2: t = turn2;
l3: (y2 == false ||  t == turn1) ;
    mutex++ ;
l4:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l1 }
}
```

```
proctype P2() {
m1: y2 = true;
m2: t = turn1;
m3: (y1 == false ||  t == turn2) ;
    mutex++ ;
m4:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m1 }
}

active proctype monitor() {
    assert (mutex <= 1)
}

init {
  atomic { run P1() ; run P2() }
}
```

Checking deadlocks

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l1: y1 = true;
l2: skip ;
l3: (y2 == false) ;
    mutex++ ;
l4:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l1 }
}
```

```
proctype P2() {
m1: y2 = true;
m2: skip ;
m3: (y1 == false) ;
    mutex++ ;
m4:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m1 }
}

active proctype monitor() {
    assert (mutex <= 1)
}

init {
  atomic { run P1() ; run P2() }
}
```

Checking progress by progress-labels

```
#define MAX 5

mtype = { mesg, ack, nak, err };

proctype sender(chan in, out)
{    byte o, s, r;
     o=MAX-1;
     do
     :: o = (o+1)%MAX; /* next msg */
again: if
       :: out!mesg(o,s) /* send */
       :: (1)  -> progress1: out!err(0,0)  /*
     distort */
       :: (1)  -> progress2: skip       /* or
     lose */
       fi;
       if
       :: timeout   -> goto again
       :: in?err(0,0) -> goto again
       :: in?nak(r,0) -> goto again
       :: in?ack(r,0) ->
            if
            :: (r == s) -> goto progress
            :: (r != s) -> goto again
            fi
       fi;
progress:   s = 1-s     /* toggle seqno */
     od
}
```

```
proctype receiver(chan in, out) {
     byte i;             /* actual input   */
     byte s;             /* actual seqno   */
     byte es;            /* expected seqno */
     byte ei;            /* expected input */
     do
     :: in?mesg(i, s) ->
            if
            :: (s == es) ->
                        assert(i == ei);
progress:               es = 1 - es;
                        ei = (ei + 1)%MAX;
                        if
     /* send,   */      :: out!ack(s,0)
     /* distort */      :: (1)  -> out!err(0,0)
                        :: (1)  -> skip
                        fi
            :: (s != es) ->
                        if
     /* send,   */      :: out!nak(s,0)
     /* distort */      :: (1)  -> out!err(0,0)
                        :: (1)  -> skip
                        fi
            fi
     :: in?err(0,0) ->  out!nak(s,0)
     od
}
```

Checking progress (ctd.)

```
init {
    chan s_r = [1] of { mtype,byte,byte };
    chan r_s = [1] of { mtype,byte,byte };
    atomic {
          run sender(r_s, s_r);
          run receiver(s_r, r_s)
    }
}
```

Progress labels: improved version

```
#define MAX 5

mtype = { mesg, ack, nak, err };

proctype sender(chan in, out)
{    byte o, s, r;
     o=MAX-1;
     do
     :: o = (o+1)%MAX; /* next msg */
again: if
       :: out!mesg(o,s) /* send */
       :: (1)  -> progress1: out!err(0,0)  /*
     distort */
       :: (1)  -> progress2: skip       /* or
     lose */
       fi;
       if
       :: timeout   -> goto again
       :: in?err(0,0) -> goto again
       :: in?nak(r,0) -> goto again
       :: in?ack(r,0) ->
            if
            :: (r == s) -> goto progress
            :: (r != s) -> goto again
            fi
       fi;
progress:   s = 1-s     /* toggle seqno */
     od
}
```

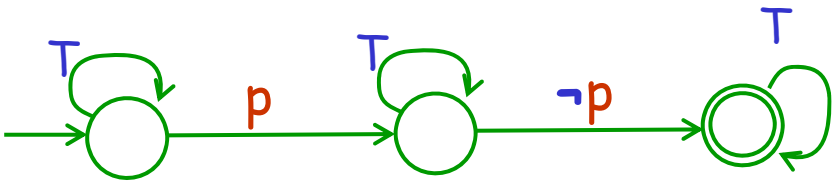
```
proctype receiver(chan in, out) {
     byte i;             /* actual input   */
     byte s;             /* actual seqno   */
     byte es;            /* expected seqno */
     byte ei;            /* expected input */
     do
     :: in?mesg(i, s) ->
            if
            :: (s == es) ->
                         assert(i == ei);
progress:                es = 1 - es;
                         ei = (ei + 1)%MAX;
                         if
     /* send,   */       :: out!ack(s,0)
     /* distort */       :: (1)  -> progress1:
     out!err(0,0)
                         :: (1)  -> progress2:
     skip
                         fi
            :: (s != es) ->
                         if
     /* send,   */       :: out!nak(s,0)
     /* distort */       :: (1)  -> progress3:
     out!err(0,0)
                         :: (1)  -> progress4:
     skip
                         fi
            fi
     :: in?err(0,0) ->  out!nak(s,0)
     od
}
```

```
init {
```

Automata properties: never claims

Automata specifications can be given in Promela as Never claims, e.g.,

- □ (p -> □ p)



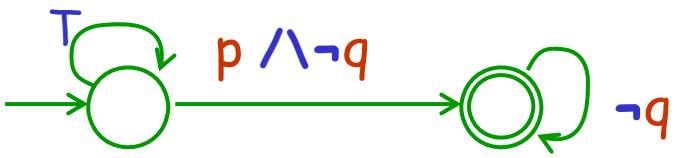

```
never {
        do
        :: skip
        :: p -> break
        od ;
        do
        :: skip
        :: !p -> break
        od
}
```

Never claims execute in lock-step with the rest of the Promela model
Accept if they reach the end

Buchi Automata as never claims

Accepting states designated by labels `acceptxxxxx`

- $\Diamond(p \wedge \Box \neg q)$




```
never {
        do
        :: skip
        :: p && !q -> break
        od ;
accept: do
        :: !q
        od
}
```

The never claim accepts if the Promela model has cycle with only `!q`

Then SPIN reports a violating cycle.