



ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

05 - Validation, Part II

Stefan Rührup


University of Freiburg
Computer Networks and Telematics
Summer 2009



Overview

- ▶ **In the first part of this chapter:**
 - Promela, a language to describe validation models

- ▶ **In this part:**
 - Model checking with SPIN
 - Example: Validation of the Alternating Bit Protocol

slides referring to this example are marked with  ABP

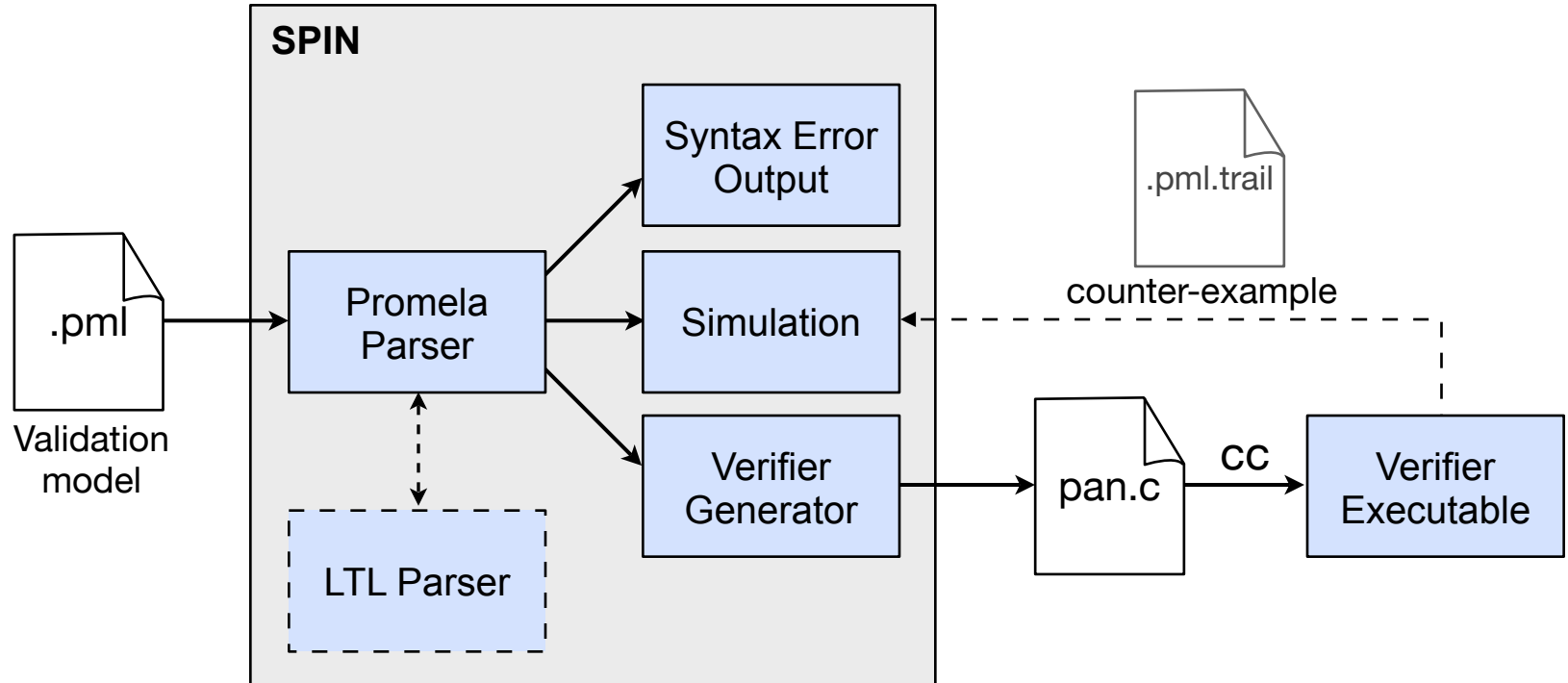
SPIN

- ▶ **SPIN Model Checker**
 - Simple Promela Interpreter
 - developed by Gerard J. Holzmann, Bell Labs
 - simulation and validation of Promela models
 - open source

- ▶ **XSpin: Tcl/Tk GUI for SPIN**

- ▶ Download: <http://spinroot.com/spin/Src/>

SPIN's Structure



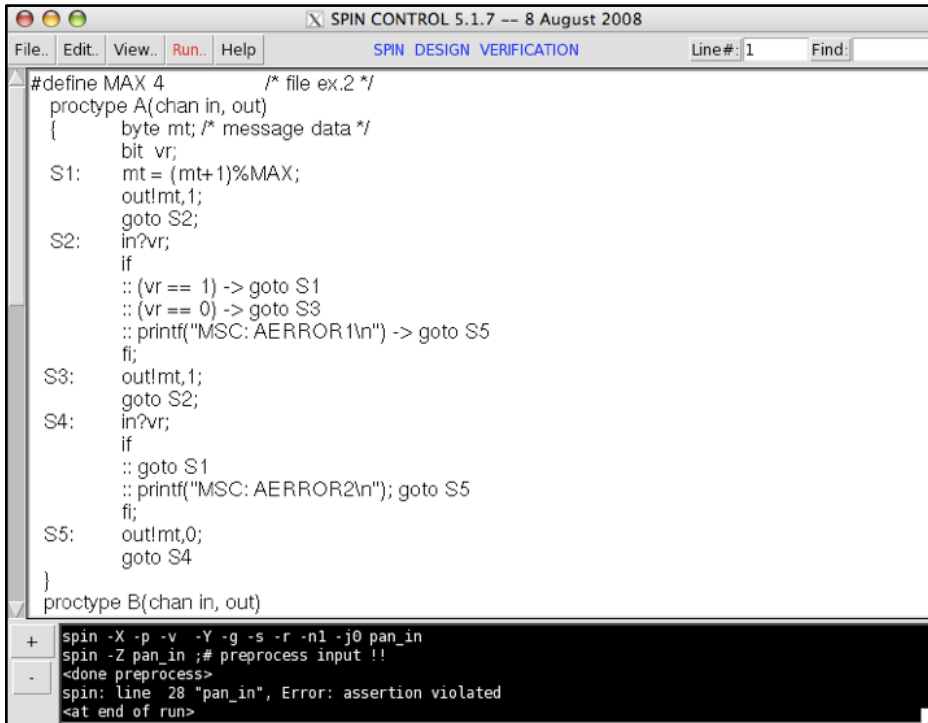
cf. [Holzmann 2003]

SPIN's Syntax

- ▶ Syntax: `spin [options] file`
- ▶ Examples:
 - > `spin -r model.pml`
- ▶ Options:
 - r print receive events
 - c produce an MSC approximation in ASCII
 - a generate analyzer
- ▶ more command line options: `spin --`
- ▶ see also <http://spinroot.com/spin/Man/Spin.html>

XSPIN

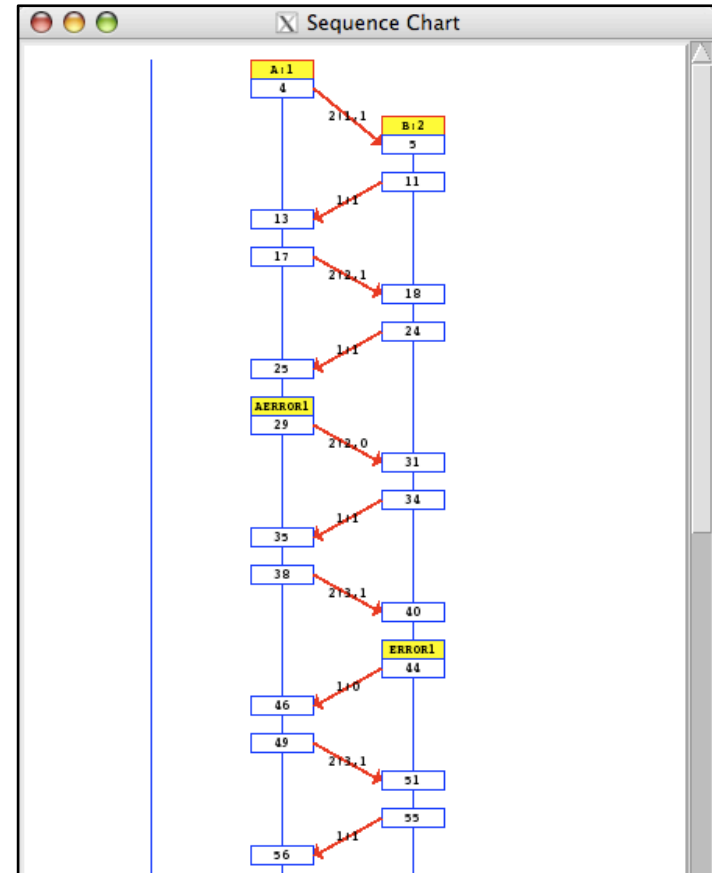
GUI for SPIN verification and simulation runs



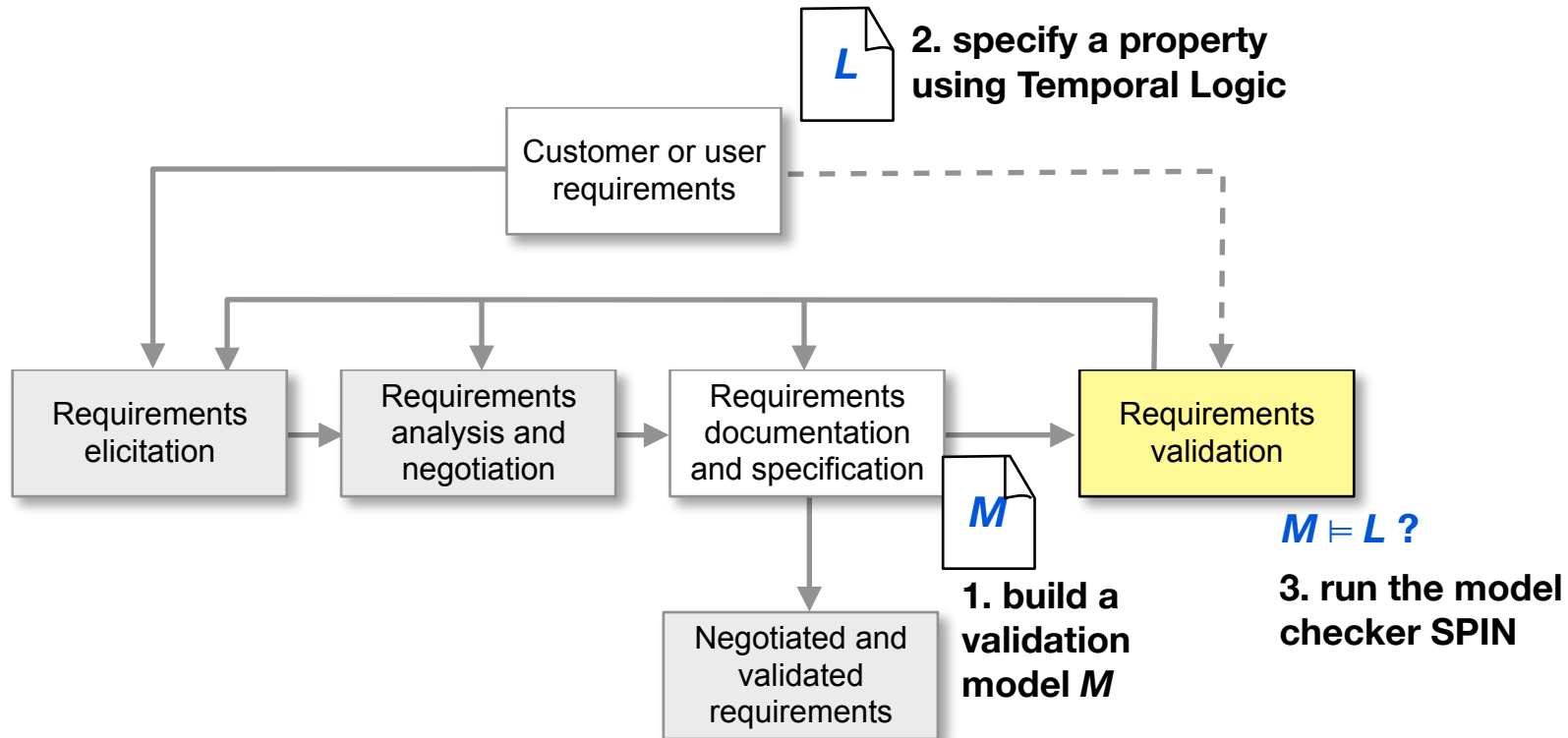
```
SPIN CONTROL 5.1.7 -- 8 August 2008
File.. Edit.. View.. Run.. Help
SPIN DESIGN VERIFICATION Line#: 1 Find:

#define MAX 4 /* file ex.2 */
proctype A(chan in, out)
{
  byte mt; /* message data */
  bit vr;
  S1: mt = (mt+1)%MAX;
  out!mt,1;
  goto S2;
  S2: in?vr;
  if
  :: (vr == 1) -> goto S1
  :: (vr == 0) -> goto S3
  :: printf("MSC: AERROR1\n") -> goto S5
  fi;
  S3: out!mt,1;
  goto S2;
  S4: in?vr;
  if
  :: goto S1
  :: printf("MSC: AERROR2\n"); goto S5
  fi;
  S5: out!mt,0;
  goto S4
}
proctype B(chan in, out)

+ spin -X -p -v -Y -g -s -r -n1 -j0 pan_in
- spin -Z pan_in ;# preprocess input !!
<done preprocess>
spin: line 28 "pan_in", Error: assertion violated
<at end of run>
```



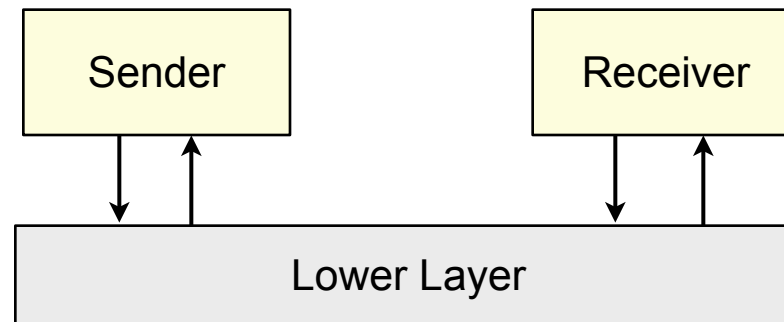
Model checking with SPIN



[S. Leue, Design of Reactive Systems, Lecture Notes, 2001]

Example: Creating a validation model

- ▶ Sender and receiver communicate over an unreliable channel (without message loss)
- ▶ Protocol: The alternating bit protocol (cf. Exercise 2)
- ▶ 3 Processes: Sender, Receiver, Lower Layer:

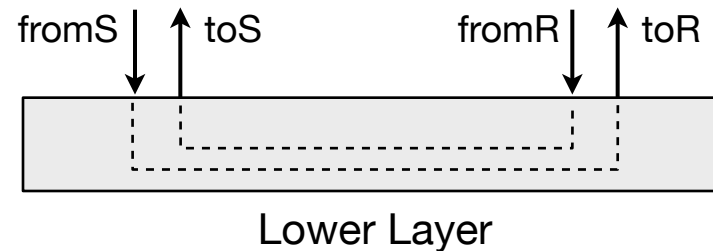


[G. J. Holzmann: “Design and validation of protocols: a tutorial”, Computer Networks and ISDN Systems, 25(9), 1993]

Modeling processes

Lower Layer model:

- ▶ Data messages are passed from the sender to the receiver.
- ▶ Acknowledgments are passed from the receiver to the sender
- ▶ Data and Acks contain an *alternating bit*



```

mtype = { data, ack }

proctype lower_layer(chan fromS, toS,
                    fromR, toR)
{
    byte d; bit b;
    do
        ::fromS?data(d,b) -> toR!data(d,b)
        ::fromR?ack(b) -> toS!ack(b)
    od
}

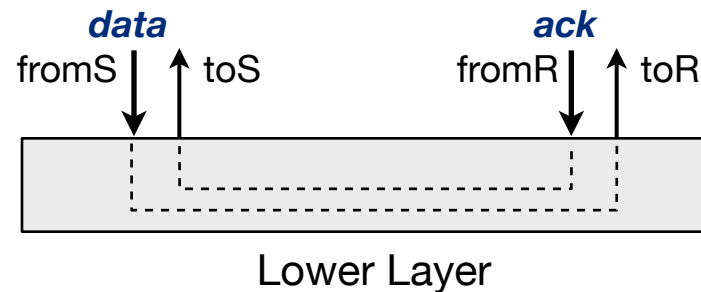
```

Modeling channels

- ▶ Channel initialization reflect the message types used here

```
#define N 2

chan fromS = [N] of { byte, byte, bit }; /* data channels */
chan toR   = [N] of { byte, byte, bit };
chan fromR = [N] of { byte, bit };      /* ack channels */
chan toS   = [N] of { byte, bit };
```



Modeling processes (cntd.)

Introducing unreliability in the lower layer:

```
proctype lower_layer(chan fromS, toS, fromR, toR)
{
  byte d; bit b;

  do
    ::fromS?data(d,b) ->
      if
        random choice → ::toR!data(d,b) /* correct */
        random choice → ::toR!error /* corrupted */
      fi
    ::fromR?ack(b) ->
      if
        ::toS!ack(b) /* correct */
        ::toS!error /* corrupted */
      fi
    od
  }
}
```

Modeling the Sender

```
proctype Sender(chan in, out)
{
  byte mt;      /* message data */
  bit at;      /* alternation bit transmitted */
  bit ar;      /* alternation bit received */

  FETCH;      /* get a new message */
  out!data(mt,at); /* ...and send it */
  do
  ::in?ack(ar) -> /* await response */
    if
      ::(ar == at) -> /* successful transmission */
        FETCH; /* get a new message */
        at=1-at /* toggle alternating bit */
      ::else -> /* there was a send error */
        skip /* don't fetch a new msg. */
    fi;
    out!data(mt,at)
  ::in?error(ar) -> /* receive error */
    out!data(mt,at) /* simply send again */
  od
}
```

Modeling the Receiver

```
proctype Receiver(chan in, out)
{
    byte mr;          /* message data received */
    byte last_mr;     /* mr of last error-free msg */
    bit ar;           /* alternation bit received */
    bit last_ar;     /* ar of last error-free msg */

    do
        ::in?error(mr,ar) ->      /* receive error */
            out!ack(last_ar);     /* send ack with old bit */
        ::in?data(mr,ar) ->
            out!ack(ar);          /* send response */
            if
                ::(ar == last_ar) -> /* bit is not alternating */
                    skip           /* ...don't accept */
                ::(ar != last_ar) -> /* bit is alternating */
                    ACCEPT;        /* correct message */
                    last_ar=ar;    /* store alternating bit */
                    last_mr=mr     /* save last message */
            fi
    od
}
```

Fetching and Accepting

- ▶ We assume that the fetched data is a sequence of integers (modulo some maximum value)

```
#define FETCH    mt = (mt+1)%MAX
```

- ▶ **Correctness claim:** The receiver should only accept those data messages that contain the correct integer value:

```
#define ACCEPT  assert(mr==(last_mr+1)%MAX)
```

Defining the initial process

```
#define N 2

init {
    chan fromS = [N] of { byte, byte, bit };
    chan toR   = [N] of { byte, byte, bit };
    chan fromR = [N] of { byte, bit };
    chan toS   = [N] of { byte, bit };

    atomic {
        run Sender(toS, fromS);
        run Receiver(toR, fromR);
        run lower_layer(fromS, toS, fromR, toR)
    }
}
```

Putting all together

```
#define N 2
#define MAX 8
#define FETCH mt = (mt+1)%MAX
#define ACCEPT assert(mr==(last_mr+1)%MAX)

mtype = { data, ack, error }

proctype lower_layer(chan fromS, toS, fromR, toR) {...}
proctype Sender(chan in, out) {...}
proctype Receiver(chan in, out) {...}

init {
    chan fromS = [N] of { byte, byte, bit };
    chan toR = [N] of { byte, byte, bit };
    chan fromR = [N] of { byte, bit };
    chan toS = [N] of { byte, bit };

    atomic {
        run Sender(toS, fromS);
        run Receiver(toR, fromR);
        run lower_layer(fromS, toS, fromR, toR) }
}
```


Running the program

- ▶ When invoking `spin filename.pml` the simulator is started.
- ▶ Simulations are random by default
- ▶ Violated assertions abort the simulation

```
> spin alternating.pml
spin: line 64 "alternating.pml", Error: assertion violated
spin: text of failed assertion: assert((mr==((last_mr+1)%8)))
#processes: 4
97: proc 3 (lower_layer) line 22 "alternating.pml" (state 10)
97: proc 2 (Receiver) line 64 "alternating.pml" (state 9)
97: proc 1 (Sender) line 33 "alternating.pml" (state 14)
97: proc 0 (:init:) line 82 "alternating.pml" (state 5) <valid end state>
4 processes created
```

Running the program again

```
> spin alternating.pml
spin: line 64 "alternating.pml", Error: assertion violated
spin: text of failed assertion: assert((mr==((last_mr+1)%8)))
#processes: 4
97: proc 3 (lower_layer) line 22 "alternating.pml" (state 10)
97: proc 2 (Receiver) line 64 "alternating.pml" (state 9)
97: proc 1 (Sender) line 33 "alternating.pml" (state 14)
97: proc 0 (:init:) line 82 "alternating.pml" (state 5) <valid end state>
4 processes created
```

This is a random simulation, let's see if the error is singular...

```
> spin alternating.pml
spin: line 64 "alternating.pml", Error: assertion violated
spin: text of failed assertion: assert((mr==((last_mr+1)%8)))
#processes: 4
34: proc 3 (lower_layer) line 18 "alternating.pml" (state 9)
34: proc 2 (Receiver) line 64 "alternating.pml" (state 9)
34: proc 1 (Sender) line 33 "alternating.pml" (state 14)
34: proc 0 (:init:) line 82 "alternating.pml" (state 5) <valid end state>
4 processes created
```

Running the program

- ▶ Before proceeding with the analysis...
Printing the message content makes life easier:

```
#define ACCEPT printf("ACCEPT %d\n", mr); assert(mr==(last_mr+1)%MAX)
```

- ▶ By choosing a fixed seed for the random simulation we obtain always the same message sequence:

```
> spin -nSEED alternating.pml
```

Showing the message sequence

```

> spin -n3 -c alternating.pm1
proc 0 = :init:
proc 1 = Sender
proc 2 = Receiver
proc 3 = lower_layer
q\p  0  1  2  3
1  .  out!3,1,0
1  .  .  .  fromS?3,1,0
2  .  .  .  toR!1,0,0
2  .  .  in?1,0,0
3  .  .  out!2,0
3  .  .  .  fromR?2,0
4  .  .  .  toS!1,0
4  .  in?1,0
...
...
3  .  .  out!2,1
3  .  .  .  fromR?2,1
4  .  .  .  toS!1,0
4  .  in?1,0
1  .  out!3,2,1
      ACCEPT 2
spin: line 64 "alternating.pm1", Error: assertion violated
spin: text of failed assertion: assert((mr==(last_mr+1)%8))

```

-c = columnated output

Showing receive events

-r = print receive events

```

> spin -n3 -r alternating.pml
 6: proc 3 (lower_layer) line 12 "alternating.pml" Recv 3,1,0 <- queue 1 (fromS)
 9: proc 2 (Receiver) line 56 "alternating.pml" Recv 1,0,0 <- queue 2 (in)
14: proc 3 (lower_layer) line 17 "alternating.pml" Recv 2,0 <- queue 3 (fromR)
18: proc 1 (Sender) line 43 "alternating.pml" Recv 1,0 <- queue 4 (in)
21: proc 3 (lower_layer) line 12 "alternating.pml" Recv 3,1,0 <- queue 1 (fromS)
24: proc 2 (Receiver) line 56 "alternating.pml" Recv 1,0,0 <- queue 2 (in)
27: proc 3 (lower_layer) line 17 "alternating.pml" Recv 2,0 <- queue 3 (fromR)
29: proc 1 (Sender) line 34 "alternating.pml" Recv 2,0 <- queue 4 (in)
39: proc 3 (lower_layer) line 12 "alternating.pml" Recv 3,2,1 <- queue 1 (fromS)
41: proc 2 (Receiver) line 56 "alternating.pml" Recv 1,0,0 <- queue 2 (in)
46: proc 3 (lower_layer) line 17 "alternating.pml" Recv 2,0 <- queue 3 (fromR)
48: proc 1 (Sender) line 34 "alternating.pml" Recv 2,0 <- queue 4 (in)
55: proc 3 (lower_layer) line 12 "alternating.pml" Recv 3,2,1 <- queue 1 (fromS)
60: proc 2 (Receiver) line 58 "alternating.pml" Recv 3,2,1 <- queue 2 (in)
62: proc 3 (lower_layer) line 17 "alternating.pml" Recv 2,1 <- queue 3 (fromR)
66: proc 1 (Sender) line 43 "alternating.pml" Recv 1,0 <- queue 4 (in)
    ACCEPT 2
spin: line 64 "alternating.pml", Error: assertion violated
spin: text of failed assertion: assert((mr==(last_mr+1)%8))


```

What is the error?

- ▶ The first accepted message contains “2”.
- ▶ Where is the first message?
- ▶ Initialization problem: `last_ar == ar` in the first round

```
proctype Receiver(chan in, out)
{
  byte mr;          /* message data received */
  byte last_mr;    /* mr of last error-free msg */
  bit ar;          /* alternation bit received */
  bit last_ar;    /* ar of last error-free msg */


  do
    ::in?error(mr,ar) ->
      out!ack(last_ar);
    ::in?data(mr,ar) ->
      ...
}
```



Running the program again

```
proctype Receiver(chan in, out)
{
  byte mr;          /* message data received */
  byte last_mr;    /* mr of last error-free msg */
  bit ar;          /* alternation bit received */
  bit last_ar=1;   /* ar of last error-free msg */

  do
    ::in?error(mr,ar) ->
      out!ack(last_ar);
    ::in?data(mr,ar) ->
      ...
}
```



- ▶ Now the simulation runs without termination ...

Verification

- ▶ The protocol runs in our random simulations.
- ▶ But does it work correctly in **all** situations?
To be checked by the verifier
- ▶ Generating and invoking a verifier with SPIN:
 - > ./spin -a alternating.pml
 - > cc pan.c -o pan
 - > ./pan

Verification of the Example

```
> ./pan
pan: too few parameters in send stmt (at depth 86)
pan: wrote alternating.pml.trail
```

The verifier is stricter than the interpreter...

```
proctype lower_layer(chan fromS, toS, fromR, toR)
{
  byte d; bit b;

  do
    ::fromS?data(d,b) ->
      if
        ::toR!data(d,b) /* correct */
        ::toR!error(0,0) /* corrupted */
      fi
    ::fromR?ack(b) ->
      if
        ::toS!ack(b) /* correct */
        ::toS!error(0) /* corrupted */
      fi
  od
}
```

Verification, again...

```

> ./pan
(Spin Version 5.1.7 -- 23 December 2008)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  acceptance cycles    - (not selected)
  invalid end states   +

State-vector 88 byte, depth reached 127, errors: 0
  510 states, stored
  139 states, matched
  649 transitions (= stored+matched)
  2 atomic steps
hash conflicts:          0 (resolved)

  2.501 memory usage (Mbyte)

unreached in proctype lower_layer
  line 23, state 14, "-end-"
  (1 of 14 states)
unreached in proctype Sender
  line 46, state 17, "-end-"
  (1 of 17 states)
unreached in proctype Receiver
  line 69, state 17, "-end-"
  (1 of 17 states)
unreached in proctype :init:
  (0 of 5 states)

```

some unreached end states,
but this is ok as the protocol
should keep on transmitting

Correctness claims

▶ Types of claims

- Safety: set of properties that the system may not violate
- Liveness: set of properties that the system must satisfy

- Reachable and unreachable states (state properties)
- Feasible and infeasible executions (path properties)

- System invariant: holds in every reachable state
- Process assertion: holds only in specific reachable states

[Holzmann 2003]

Safety and Liveness

Safety “something bad never happens” Properties that the system may not violate	Liveness “something good will eventually happen” Properties that the system must satisfy
Definition of valid states No assertions are violated There are no deadlocks (invalid end states)	Progress is enforced There are no livelocks (non-progress cycles)
Verification: Show that there is no trace leading to the “bad” things (deadlocks, violated invariants, ...)	Verification: Show that there is no (infinite) loop in which the “good” things do not happen

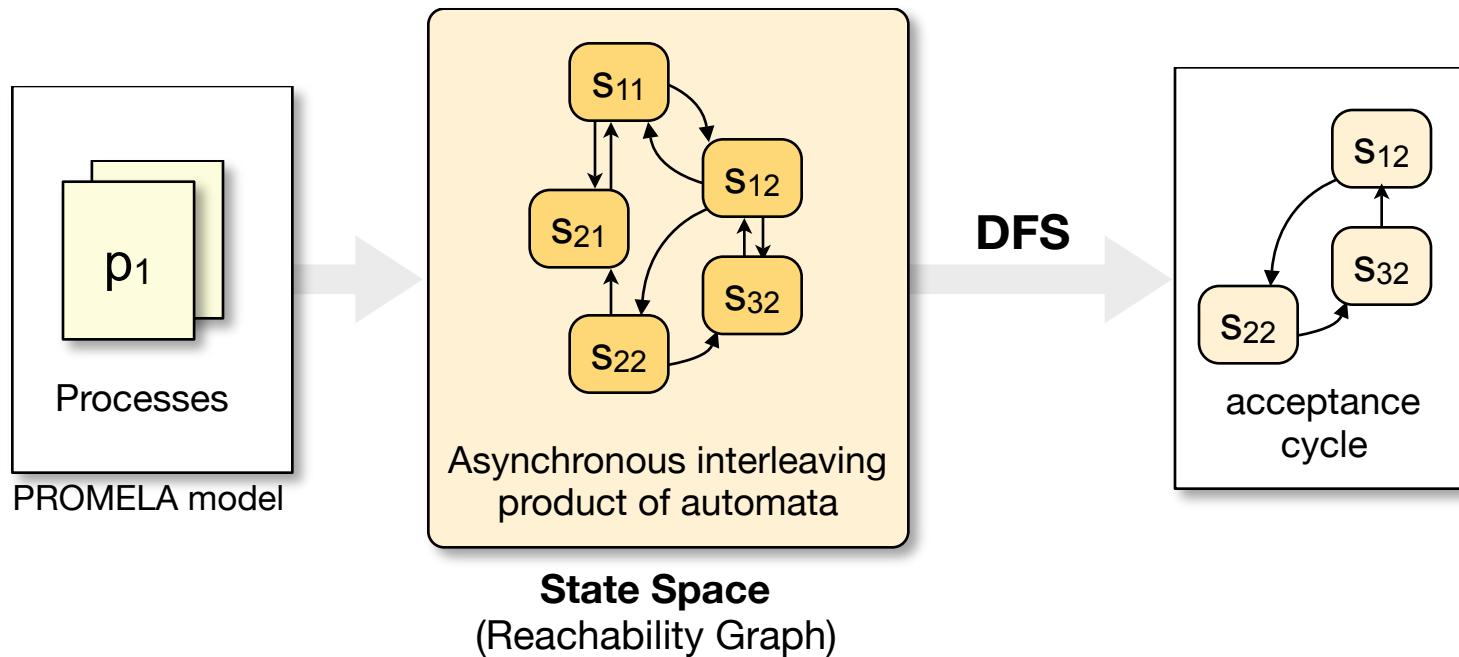
cf. [T.C. Ruys, Spin Tutorial, 2004]

Correctness properties in Promela

- ▶ Basic assertions
- ▶ Meta-Labels for identifying
 - End states
 - Progress states
 - Accept states
- ▶ Never claims
 - ... for defining safety and liveness properties
- ▶ Trace assertions
 - ... for defining properties of message channels

[Holzmann 2003]

How SPIN checks correctness



[G.J. Holzmann: "The Model Checker SPIN", IEEE Transactions on Software Engineering, 23(5), 1997]

Checking cycles and fairness

- ▶ SPIN checks for deadlocks and non-progress cycles
- ▶ There is no way to define relative speed
- ▶ Isn't it then possible that one process is infinitely slow and another one infinitely fast?
... and the slow process will never be able to execute the next statement?
- ▶ Therefore SPIN allows to check the model under the assumption of *fairness*.

Fairness (1)

- ▶ There is no assumption about relative execution speed, thus infinite delays are possible
- ▶ A fair treatment of the processes in their execution is expressed by the assumption of *finite progress*
- ▶ **Any process that can execute a statement will eventually proceed in executing it.**
- ▶ SPIN supports two variants ...

[Holzmann 2003]

Fairness (2)

- ▶ **Weak Fairness**

If a process has an executable statement whose executability never changes, then it will eventually execute that statement

- ▶ **Strong Fairness**

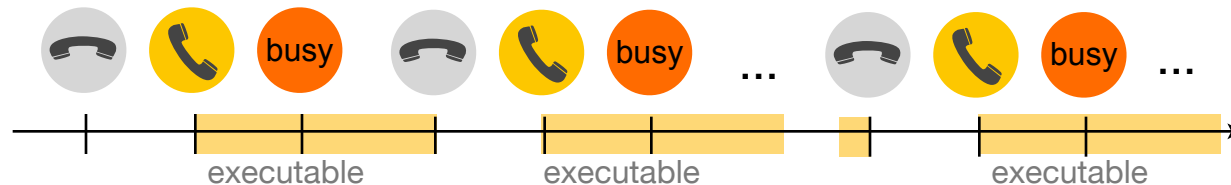
If a process has a statement that becomes executable infinitely often, then it will eventually execute that statement

[Holzmann 2003]

Fairness (3)

- ▶ **Example:** A caller picks up the receiver, dials a number (phone call gets executable), the line is busy, the caller hangs up (phone call is not executable)

Weak fairness: he does not need to be served



Strong fairness: he is eventually being served



Assertions

- ▶ **Basic assertions** `assert(expression)`
 - always executable, violation triggers an error message
 - can be used in simulation mode (abort on error)

- ▶ **Channel assertions**
 - Exclusive send and exclusive receive

```
proctype Sender(...) {
    xs ch1;    /* only Sender sends to channel ch1 */
    xr ch2;    /* only Sender receives from channel ch2 */
    ...
}
```
 - Validity of `xs`, `xr` is checked during verification.

End state labels

- ▶ Labels with the prefix `end` mark a valid end state
- ▶ Default end states: end of the process code
- ▶ End state labels enable the verifier to distinguish between valid and invalid end states
- ▶ By default, SPIN (in verification mode) checks for invalid end states
- ▶ Strict check (`spin -q`): A system state is valid, if all processes have reached a valid end state *and* all message queues are empty.

Progress state labels

- ▶ When is a cyclic execution valid?
- ▶ Statements that constitute a progress can be labeled with progress state labels.
- ▶ Progress state labels have the prefix `progress`
- ▶ Enabling non-progress checking in the verifier:

```
cc -DNP pan.c -o pan  
./pan -1
```
- ▶ Compiler flag `-DNP` lets SPIN generate a so-called never claim that checks the non-progress property

Accept state labels

- ▶ Accept states are states that should not be passed through infinitely often.
- ▶ Usually used in never claims
- ▶ Cycles passing through an accept state will be reported as an error by the verifier.

Example: Dijkstra's Semaphore

```
public class Semaphore {
    private int count = 1; ← number of permits, here only 1

    public Semaphore(int count) {
        // if (count > 1) this.count = count; ← binary version
        // (mutex)
    }

    public synchronized void P() { ← probeer te verlagen
        while (count <= 0)           (try to decrease)
        {
            try {
                wait();
            } catch( InterruptedException e ) {}
            count--;
        }

    public synchronized void V() ← verhogen (increase)
    {
        count++;
        notify();
    }
}
```

Semaphore.java

The Semaphore in Promela

```
mtype {p,v}

chan sema = [0] of {mtype}

active proctype Semaphore() {
  do
    :: sema!p -> sema?v
  od
}

active [3] proctype user() {
  do
    :: sema?p; /* enter critical section */
    skip; /* critical section */
    sema!v; /* leave critical section */
  od
}
```

semaphore.pml

cf. [Holzmann 1991]

Correctness of the semaphore

- ▶ Safety and liveness properties of the semaphore algorithm
- ▶ **Safety:** Only one process is in its critical section at any time
- ▶ **Liveness:** Whenever a process wants to enter its critical section, it will eventually be permitted to do so.
 - Liveness check: searching for non-progress cycles

Liveness check

```
> spin -a semaphore.pml
> cc -DNP pan.c -o pan
> ./pan -l
pan: non-progress cycle (at depth 3)
pan: wrote semaphore.pml.trail

(Spin Version 5.1.7 -- 23 December 2008)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
never claim          +
assertion violations + (if within scope of claim)
non-progress cycles  + (fairness disabled)
invalid end states  - (disabled by never claim)

State-vector 36 byte, depth reached 10, errors: 1
    4 states, stored
    0 states, matched
    4 transitions (= stored+matched)
    0 atomic steps
hash conflicts:      0 (resolved)
```

Guided simulation

```
> spin -t -p semaphore.pm1 (SPIN uses the recorded trail here)
Starting Semaphore with pid 0
Starting user with pid 1
Starting user with pid 2
Starting user with pid 3
spin: couldn't find claim (ignored)
  2: proc 0 (Semaphore) line 7 "semaphore.pm1" (state 1) [sema!p]
  3: proc 3 (user) line 16 "semaphore.pm1" (state 1) [sema?p]
  <<<<<START OF CYCLE>>>>>
  5: proc 3 (user) line 17 "semaphore.pm1" (state 2) [(1)]
  7: proc 3 (user) line 18 "semaphore.pm1" (state 3) [sema!v]
  8: proc 0 (Semaphore) line 8 "semaphore.pm1" (state 2) [sema?v]
 10: proc 0 (Semaphore) line 7 "semaphore.pm1" (state 1) [sema!p]
 11: proc 3 (user) line 16 "semaphore.pm1" (state 1) [sema?p]
spin: trail ends after 11 steps
#processes: 4
 11: proc 3 (user) line 17 "semaphore.pm1" (state 2)
 11: proc 2 (user) line 15 "semaphore.pm1" (state 4)
 11: proc 1 (user) line 15 "semaphore.pm1" (state 4)
 11: proc 0 (Semaphore) line 8 "semaphore.pm1" (state 2)
4 processes created
```

considered
as non-
progress
cycle

Adding labels

we mark this as
progress state

```
...
active proctype Semaphore() {
end:      do
          :: sema!p ->
progress:   sema?v
          od
}
...
```

semaphore.pml

```
> ./pan -l
(Spin Version 5.1.7 -- 23 December 2008) ... no more error messages
+ Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations + (if within scope of claim)
  non-progress cycles  + (fairness disabled)
  invalid end states   - (disabled by never claim)
...

```

100% free from assertion
violations and non-prog. cycles

Never claims

- ▶ **Expressing temporal claims**
e.g. “every system state in which P is true is followed by a system state in which Q is true”
- ▶ Notation: `never { ... }`
- ▶ The never process is executed at each step
- ▶ If the specified condition is matching and the never process reaches an end state, the claim is violated and an error is reported

Never claims, Example

- ▶ **Checking whether a property p is true**
- ▶ p should never fail:

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

- ▶ As long as p is true the never process stays in its do-loop

[Holzmann 2003]

Never claims, Example

- ▶ **Checking whether a property p is true**
- ▶ Alternative solutions: With an assertion:

```
never {  
  do  
  :: assert(p)  
  od  
}
```

... or as a separate proctype

```
active proctype monitor()  
{  
  atomic { !p -> assert(false) }  
}
```

[Holzmann 2003]

Validation of ABP (cntd.)

- ▶ Correctness of the Alternating Bit Protocol:
 - Every message is received at least once
 - Every message is accepted at most once(see also Exercise 2)
- ▶ 2nd claim already shown by using an assertion:
`#define ACCEPT assert(mr==(last_mr+1)%MAX)`
- ▶ We try to express the first claim in Promela
(though it was already implied by the last validation)
- ▶ But first, we check for non-progress cycles

[Holzmann 1993]

Non-progress loops

- ▶ The execution sequences in ABP are cyclic and by default considered to be non-progress cycles

```
> spin -a alternating.pml
> cc pan.c -DNP -o pan
> ./pan -l
pan: non-progress cycle (at depth 14)
pan: wrote alternating.pml.trail

(Spin Version 5.1.7 -- 23 December 2008)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          +
assertion violations + (if within scope of claim)
non-progress cycles  + (fairness disabled)
invalid end states   - (disabled by never claim)
```

Adding Labels

```

proctype Receiver(chan in, out)
{
  byte mr;          /* message data received */
  byte last_mr;    /* mr of last error-free msg */
  bit ar;          /* alternation bit received */
  bit last_ar;     /* ar of last error-free msg */

  do
    ::in?error(mr,ar) -> /* receive error */
      out!ack(last_ar); /* send ack with old bit */
    ::in?data(mr,ar) ->
      out!ack(ar);      /* send response */
      if
        ::(ar == last_ar) -> /* bit is not alternating */
          skip /* ...don't accept */
        ::(ar != last_ar) -> /* bit is alternating */
          progress: ACCEPT; /* correct message */
          last_ar=ar; /* store alternating bit */
          last_mr=mr /* save last message */
      fi
    od
}

```

Accepting a message is clearly a progress



Disappointing...

- ▶ There are still non-progress loops

```
> spin -a alternating.pml ; cc pan.c -DNP -o pan
> ./pan -1
pan: non-progress cycle (at depth 22)
pan: wrote alternating.pml.trail

...
```

- ▶ We will have a look at the trail
`spin -t -p alternating.pml`

The trail

```

> >spin -t -p alternating.pml
Starting :init: with pid 0
spin: couldn't find claim (ignored)
Starting Sender with pid 2
  2: proc 0 (:init:) line 78 "alternating.pml" (state 1) [(run Sender(toS,fromS))]
Starting Receiver with pid 3
  3: proc 0 (:init:) line 79 "alternating.pml" (state 2) [(run Receiver(toR,fromR))]
Starting lower_layer with pid 4
  4: proc 0 (:init:) line 80 "alternating.pml" (state 3) [(run lower_layer(fromS,toS,fromR,toR))]
  6: proc 1 (Sender) line 31 "alternating.pml" (state 1) [mt = ((mt+1)%8)]
  8: proc 1 (Sender) line 32 "alternating.pml" (state 2) [out!data,mt,at]
 10: proc 3 (lower_layer) line 12 "alternating.pml" (state 1) [fromS?data,d,b]
 12: proc 3 (lower_layer) line 15 "alternating.pml" (state 3) [toR!error,0,0]
 14: proc 2 (Receiver) line 56 "alternating.pml" (state 1) [in?error,mr,ar]
 16: proc 2 (Receiver) line 57 "alternating.pml" (state 2) [out!ack,last_ar]
 18: proc 3 (lower_layer) line 17 "alternating.pml" (state 6) [fromR?ack,b]
 20: proc 3 (lower_layer) line 19 "alternating.pml" (state 7) [toS!ack,b]
 22: proc 1 (Sender) line 34 "alternating.pml" (state 3) [in?ack,ar]
  <<<<<START OF CYCLE>>>>
 24: proc 1 (Sender) line 39 "alternating.pml" (state 7) [else]
 26: proc 1 (Sender) line 40 "alternating.pml" (state 8) [(1)]
 28: proc 1 (Sender) line 42 "alternating.pml" (state 11) [out!data,mt,at]
 30: proc 3 (lower_layer) line 12 "alternating.pml" (state 1) [fromS?data,d,b]
 32: proc 3 (lower_layer) line 15 "alternating.pml" (state 3) [toR!error,0,0]
 34: proc 2 (Receiver) line 56 "alternating.pml" (state 1) [in?error,mr,ar]
 36: proc 2 (Receiver) line 57 "alternating.pml" (state 2) [out!ack,last_ar]
 38: proc 3 (lower_layer) line 17 "alternating.pml" (state 6) [fromR?ack,b]
 40: proc 3 (lower_layer) line 19 "alternating.pml" (state 7) [toS!ack,b]
 42: proc 1 (Sender) line 34 "alternating.pml" (state 3) [in?ack,ar]
spin: trail ends after 42 steps

```

no ACCEPT here ←

Labeling lower layer progress

- ▶ Distorting messages by the lower layer can lead to cycles
- ▶ We mark this as progress as well

```

proctype lower_layer(chan fromS, toS, fromR, toR)
{
  byte d; bit b;

  do
    ::fromS?data(d,b) ->
progress0:  if
                :: toR!data(d,b)
                :: toR!error(0,0)
              fi
    ::fromR?ack(b) ->
progress1:  if
                :: toS!ack(b)
                :: toS!error(0)
              fi
    od
}

```

Message distortion is not desired, it is only marked as a normal behaviour!

- ▶ Finally, SPIN does not detect non-progress cycles any more

Specifying a never claim

- ▶ To show: Every message is received at least once
- ▶ There is no infinite sequence of duplicate messages unless they were distorted

- ▶ Therefore this should never happen:

The Receiver reaches the state of detecting a duplicate and visits this state again without having accepted a valid message.

If there was such a cycle, the receiver would have no chance to receive a valid message afterwards

Labels used in the claim

```

proctype Receiver(chan in, out)
{
  byte mr;          /* message data received */
  byte last_mr;    /* mr of last error-free msg */
  bit ar;          /* alternation bit received */
  bit last_ar;     /* ar of last error-free msg */

  do
    ::in?error(mr,ar) ->      /* receive error */
      out!ack(last_ar);     /* send ack with old bit */
    ::in?data(mr,ar) ->
      out!ack(ar);          /* send response */
      if
        ::(ar == last_ar) -> /* bit is not alternating */
          dup: skip        /* ...don't accept */
        ::(ar != last_ar) -> /* bit is alternating */
          progress: ACCEPT; /* correct message */
                      last_ar=ar; /* store alternating bit */
                      last_mr=mr  /* save last message */
      fi
    od
}

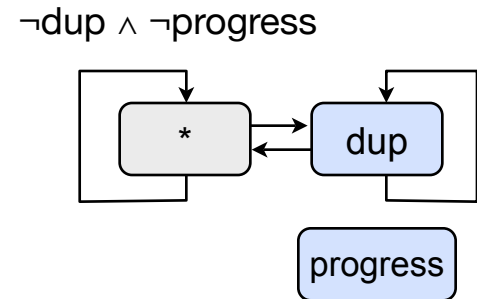
```

duplicate
received →

→ correct msg.
received

The Never Claim (1)

- ▶ Never: *The Receiver reaches the state of detecting a duplicate and visits this state again without having accepted a valid message.*



```

never {
accept: do
  :: do
  :: !Receiver@dup && !Receiver@progress
  :: Receiver@dup -> break
  od;
  :: do
  :: Receiver@dup
  :: !Receiver@dup && !Receiver@progress -> break
  od
od
}
  
```

as long as the Receiver is in other states: stay in the loop

switch to the 2nd part of the claim

The Receiver leaves the dup state without visiting the progress state

The Never Claim (2)

- ▶ Unfortunately this gives an error:

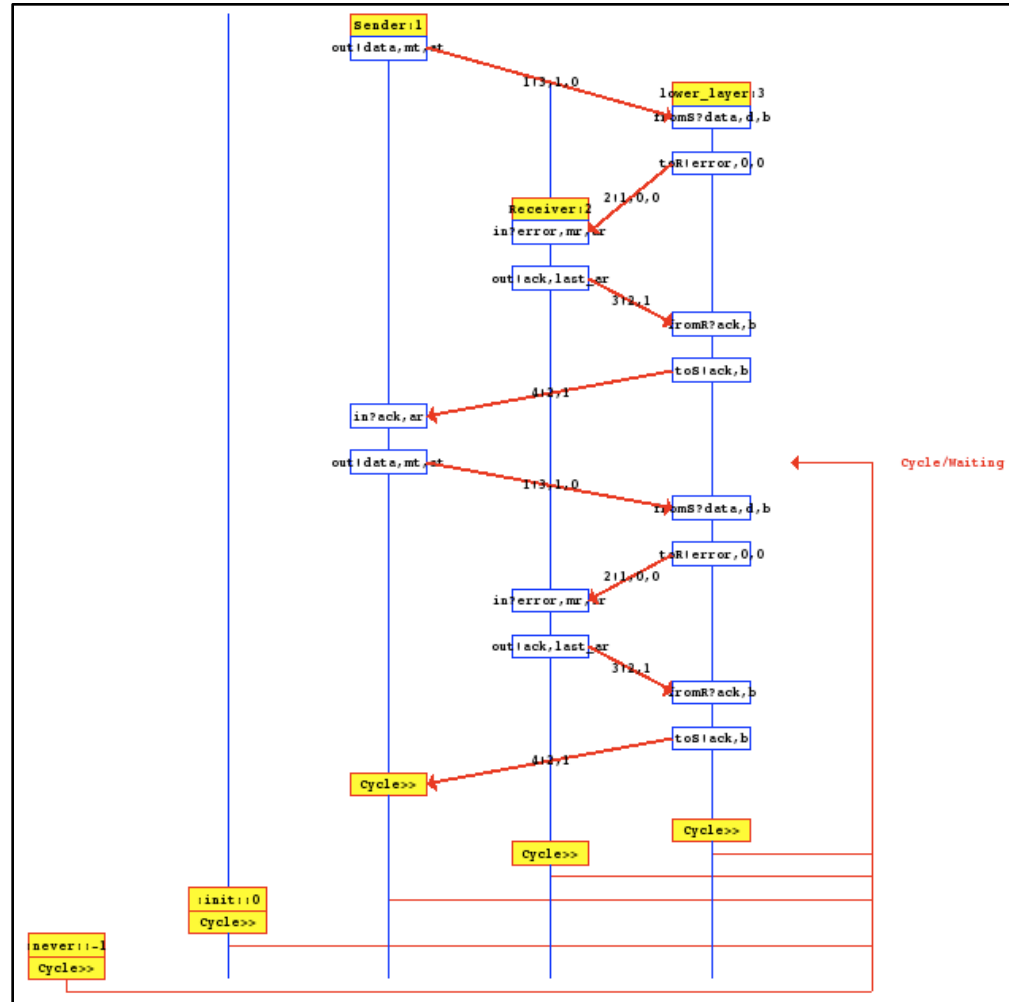
```
> spin -a alternating.pml ; cc pan.c -o pan -DNOREDUCE
> ./pan -a
pan: acceptance cycle (at depth 22)
pan: wrote alternating.pml.trail
```

- ▶ A look at the trail shows the reason: Distorting messages can lead to repeated duplicates

```
<<<<<START OF CYCLE>>>>>
24: proc 1 (Sender) line 41 "alternating.pml" (state 7) [else]
26: proc 1 (Sender) line 42 "alternating.pml" (state 8) [(1)]
28: proc 1 (Sender) line 44 "alternating.pml" (state 11) [out!data,mt,at]
30: proc 3 (lower_layer) line 12 "alternating.pml" (state 1) [fromS?data,d,b]
32: proc 3 (lower_layer) line 16 "alternating.pml" (state 3) [toR!error,0,0]
34: proc 2 (Receiver) line 58 "alternating.pml" (state 1) [in?error,mr,ar]
36: proc 2 (Receiver) line 59 "alternating.pml" (state 2) [out!ack,last_ar]
38: proc 3 (lower_layer) line 18 "alternating.pml" (state 6) [fromR?ack,b]
40: proc 3 (lower_layer) line 21 "alternating.pml" (state 7) [toS!ack,b]
42: proc 1 (Sender) line 36 "alternating.pml" (state 3) [in?ack,ar]
```

The Never Claim (3)

Graphical representation:
The acceptance cycle
in the message sequence
chart generated by XSPIN



The Never Claim (4)

- ▶ Never: *The Receiver reaches the state of detecting a duplicate and visits this state again without having accepted a valid message ... unless there was an error*

```
never {
accept: do
    :: do
        :: !Receiver@dup && !Receiver@progress0
            && !lower_layer@progress0
        :: Receiver@dup -> break
    od;
    :: do
        :: Receiver@dup
        :: !Receiver@dup && !Receiver@progress0
            && !lower_layer@progress0 -> break
    od
od
}
```

Reference to the process state

Remote Referencing

- ▶ References to process state labels and variables are needed in never claims
- ▶ Reference to a process state:
`procname[pid]@label`
- ▶ Reference to a local variable:
`procname[pid]:variable`
- ▶ `pid` = process ID (instantiation number), can be omitted if there is only a single instance of a proctype.

Predefined variables and functions

Value or Function	Description	Application
<code>_pid</code> _ (underscore)	Process ID of the local process global write-only variable, used for scratch values	used in proctype declarations
<code>_np</code> <code>_last</code>	true, iff the system is in a non-progress state (all processes are currently not in a progress state) PID of the process that executed the last step	used in never claims
<code>pc_value(pid)</code> <code>enabled(pid)</code>	internal state number of the currently active process true iff the current process has an executable statement	used in never claims

Check for non-progress loops

- ▶ SPIN's built-in check for non-progress loops uses a never claim using the `_np` variable:

```
never { /* non-progress: <>[] _np */
    do
        :: skip
        :: _np -> break
    od;
accept: do
        :: _np
    od
}
```

Note on never claims

- ▶ Temporal conditions in the never claim must be free of side effects
 - no assignments
 - no receive or send operations
- ▶ The never process *monitors* system behavior

Trace Assertions

- ▶ Trace assertions describe correctness properties of message channels. They apply only to send and receive operations on message channels.

- ▶ Example:

```
trace { do
        :: out!data; in?ack
      od }
```

This assertion specifies that send and receive events are alternating and messages are of type *data* and *ack*.

- ▶ Trace assertions are used to specify valid event sequences
- ▶ Only simple send and receive operations are allowed

[Holzmann 2003]

Trace Assertions and Never Claims

Never Claim	Trace Assertion
Specifies invalid system states	Specifies event sequences
Monitors system states globally	Monitors a subset of events
Executed synchronously with the system	Executed only if monitored events occur
Can be non-deterministic	Must be deterministic

[Holzmann 2003]

Overview of correctness claims

Type of claim	Correctness property
Assertion (statement)	the specified expression must not evaluate to false
End state label	the system must not terminate unless all processes have terminated or stopped at one of the labeled end states
Progress label	the system must not execute forever <i>without</i> visiting at least one of the labeled progress states infinitely often
Accept state label	the system must not execute forever <i>while</i> visiting at least one of the labeled accept states infinitely often
Never claim	the system must not show behavior specified in the claim
Trace assertion	the system must not produce event traces other than specified

[Holzmann 2003]

Lessons learned

- ▶ Validation includes checks for different properties (absence of deadlocks, non-progress loops, ...)
- ▶ Basic correctness properties can be expressed by assertions and special labels in Promela (easy to define, efficiently checkable)
- ▶ Temporal claims refer to the control flow. They have to be specified in a never claim