

Modeling and Verification with Promela

Wishnu Prasetya

wishnu@cs.uu.nl

www.cs.uu.nl/docs/vakken/pv

Overview

- Promela modeling language
 - Architecture of SPIN
 - Examples of Promela models
-
- Acknowledgement: some slides are taken and adapted from Theo Ruys's SPIN Tutorials.

Spin and Promela

- Promela = *Process Meta Language*
 - Is a *modelling* language! (not a language to build an application)
- Strong features :
 - Powerful constructs to synchronize concurrent processes
 - Powerful model checking
 - Simulation to support analysis (of the models)
- SPIN = Simple *Promela Interpreter*

System, process, and action.

- A **system** in Promela consists of a set of concurrent (and interacting) processes.
- Each process in Promela is sequential, but possibly non-deterministic.
- Each process is built from **atomic** actions (transition).
- Concurrent execution of multiple processes is modeled by **interleaving** the processes atomic actions.
- Fairness can be imposed.

Example

```
byte x = 1 ;
```

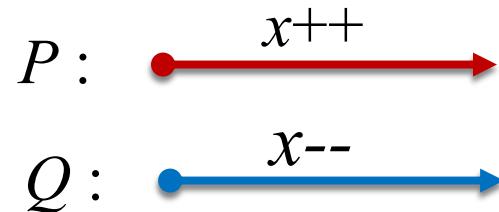
```
active proctype P() { x++ ; assert (x==2) ; }
```

```
active proctype Q() { x-- ; }
```

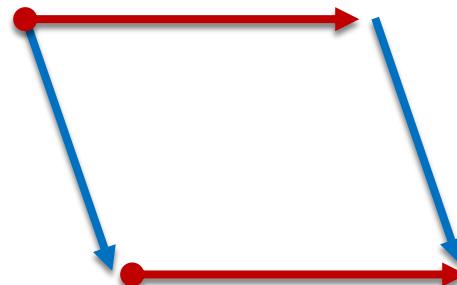
(using a global variable to interact)

Interleaving model of concurrency

- Consider (with pseudo notation), where each arrow is **atomic**:

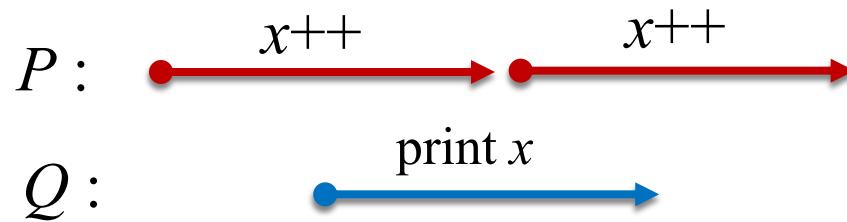


- Concurrent execution of $P||Q$ has the same effect as interleaving execution :

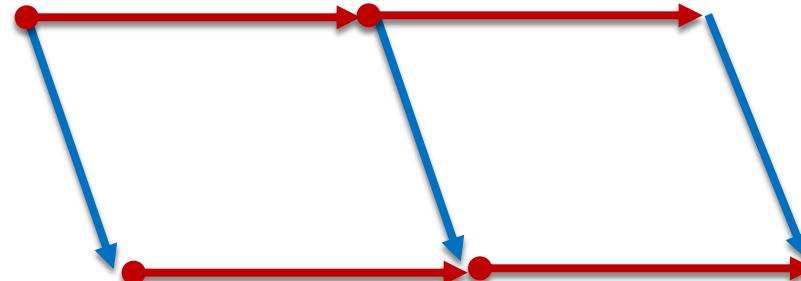


Interleaving model of concurrency

- Another example. Again each arrow is **atomic**.



- An execution of $P||Q$ abstractly proceeds as one of these paths :



Degree of atomicity

- Whether it is reasonable to model a statement as ‘atomic’, depends on your situation.
 - $x++$ usually no problem
 - $x > 0 \rightarrow y := x$ ok, if we can lock both x and y
 - $0 \in S \rightarrow found := true$?

Promela's data types

- Bit 0,1
- Bool true, false
- Byte 0..255
- Short - 2^{15} .. $2^{15}-1$
- Int - 2^{31} .. $2^{31}-1$
- Pid 0..255
- Mtype 0..255 // user-def. enumeration
- Chan 0..255
- One dimensional array
- Record

What you don't have...

- No sophisticated data types
- No methods ; you have macros
- There are only **2** levels of scope:
 - global var (visible in the entire sys)
 - local var (visible only to the process that contains the declaration)
 - there is no inner blocks

(Enabledness) Expression

```
active proctype P { x++ ; (y==0) ; x-- }
```

- This process has 3 atomic actions.
- The action “y==0”
 - only enabled in a state where the expression is true
 - it can only be executed when it is enabled; the effect is skip
 - so, as long as it is disabled, the process will block
 - if it is not enabled in the current state, a transition in another process may make it enabled in the next state.
 - even if it is enabled in the current state, there is no guarantee the action will be selected for execution; but there is a way in SPIN to impose fairness.

Example

- Use it to synchronize between processes :

```
byte x=0 , y=0
```

```
active proctype P { x++ ; (y>0) ; x-- }
```

```
active proctype Q { (x>0) ; y++ ; (x==0) ; y-- }
```

- // both will terminate, but forcing Q to finish last

Concurrent programming is tricky....

- E.g. one or more processes can become stuck (deadlocked) :

```
byte x=0 , y=0
```

```
active proctype P { x++ ; (y>0) ; x-- ; (y==0) }
```

```
active proctype Q { y++ ; (x>0) ; (x==0) ; y-- }
```

(6 potential executions...)

Processes can also synchronize with channels

```
chan c = [3] of {byte} ;  
  
active proctype producer() {  
    do  
        :: c ! 0  
    od  
}  
  
active proctype consumer() {  
    do  
        :: c ? x  
    od  
}
```

Channels

mtype = { DATA, ack }

```
chan c      = [0] of {bit};  
chan d      = [2] of {mtype, bit, byte};  
chan e[2]    = [1] of {bit};
```

- for exchanging messages between processes
- finite sized and asynchronously, unless you set it to size 0 → synchronous channel
- Syntax :

c ! 0 sending over channel c; blocking if c is full

c ? x receives from c, transfer it to x; blocking if c is empty

d ? DATA, b, y match and receives

- There are some more exotic channel operations : checking empty/full, testing head-value, copying instead of receiving, sorted send, random receive ... → check out the Manual

Conditional

```
if  
:: stmt1  
:: ...  
:: stmtn  
fi
```

```
if  
:: stmt1  
:: ...  
else -> ...  
fi
```

- The alternatives do not have to be atomic!
- The first action in an alternative acts as its “guard”, which determines if the alternative is enabled on a given state.
- Non-deterministically choose one enabled alternatives.
- If there is none, the entire IF blocks.
- “else” is a special expression that is enabled if all other alternatives block.

Non-determinism can be useful for modeling

```
proctype consumer() {  
    if  
        :: c ? x ;  
        :: c ? x ; x=corrupt ; // to model occasional corrupted data  
    fi  
    do something with x  
}
```

loop : do-statement

```
do
:: stmt1
:: ...
:: stmtn
od
```

- Non-deterministic, as in IF
- If no alternative is enabled, the entire loop blocks.
- Loop on forever, as long as there are enabled alternatives when the block cycle back.
- To exit you have explicitly do a break.

Examples

do

:: { (i>0) ; i-- }
:: { (i==0) ; **break** }

do

do

:: { (i>0) ; i-- }
:: **break**

do

do

:: (i>0) → i--
:: (i==0) → **break**

do

Label and jump

```
active proctype P() {  
    (x==0) ;  
    L0:  
    if  
        :: ... goto L0 ;  
        :: ...  
    fi  
}
```

- Labels can also be useful in specification, e.g.

 $\langle\rangle P@L0$
- Referring to labels as above goes actually via a mechanism called “remote reference”, which can also be used to inspect the value of local variables for the purpose of specification.

Exception/Escape

- $S \text{ unless } E$
- Statement! Not to be confused with LTL “unless”.
- If E ever becomes enabled during the execution of S , then S is aborted and the execution continues with E .

More precisely... check manual.

Predefined variables in Promela

- `_pid` (local var) current process' instantiation number
- `_nr_pr` the number of active processes
- `np_` true when the model is *not* in a “progress state”
- `_last` the pid of process that executed last
- `else` true if no statement in the current process is executable
- `timeout` true if no statement in the system is executable
- ...

Timeout

```
do
:: c ? x → ...
:: timeout → break
od
```

- timeout
 - useful as a mechanism to avoid deadlock
 - beware of statements that are always executable.

Expressing correctness with assertions

active proctype P ...

active proctype Q { ...; **assert** (x==0 && y==0) }

(here it implies that when Q terminates, x and y should be 0)

We can also express system invariant

- Thanks to built-in non-determinism in the interleaving semantics, we can also use assertion to specify a system/global invariant !

```
byte x=0 , y=0
```

```
active proctype P { x++ ; (y>0) ; x-- ; (y==0) }
```

```
active proctype Q { y++ ; (x>0) ; (x==0) ; y-- }
```

```
active proctype Monitor { assert ((x==0 || x==1)) }
```

// implying that at any time during the run x is either 0 or 1

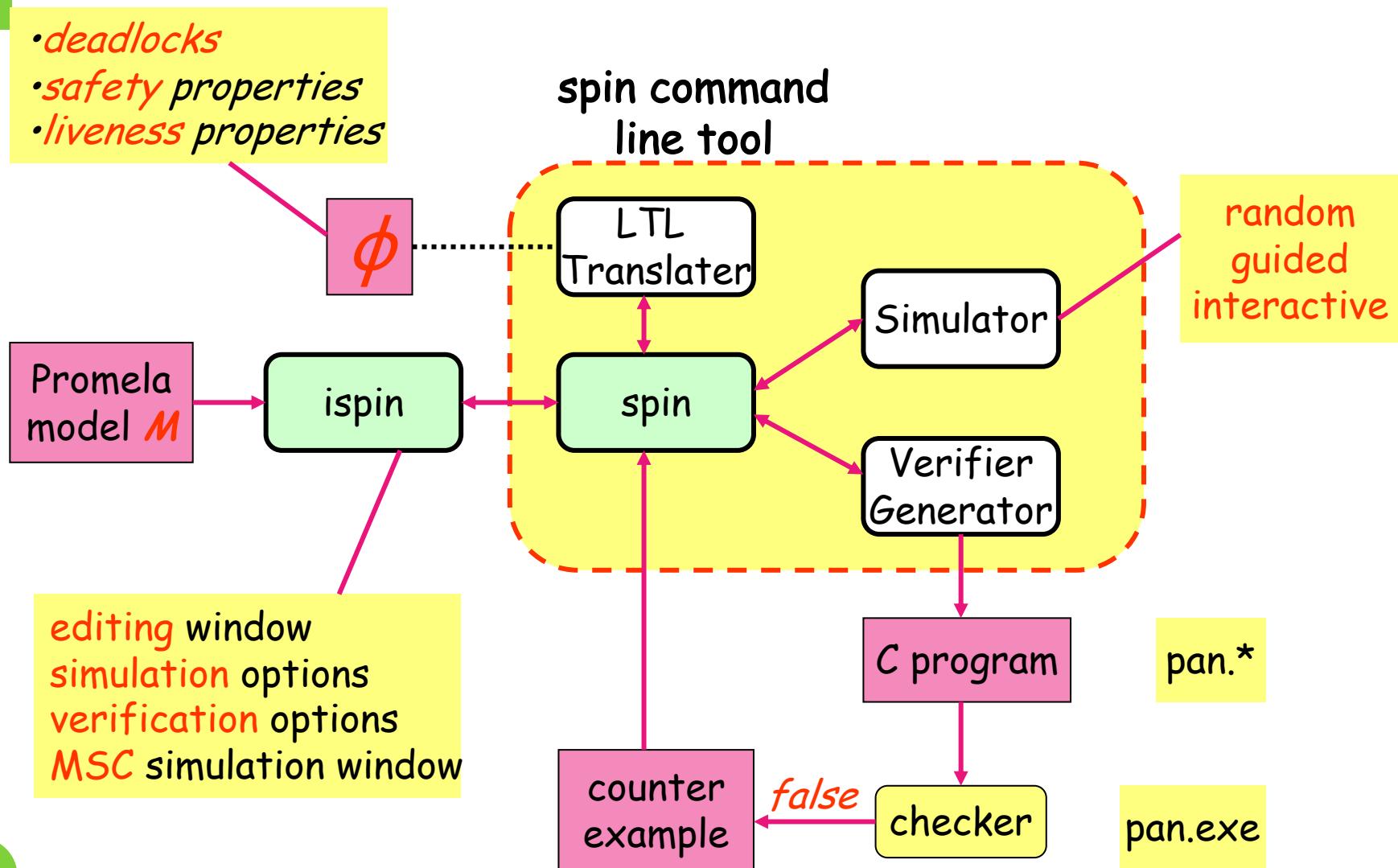
“System stuck” checking

- When a system comes to a state where it has no enabled transition, but one of its processes is not in its terminal (end) state:
 - “Invalid end state” will be reported by SPIN
 - But sometimes you want to model that this is ok → suppress it via the invalid-endstate option.
- The terminal state of a process P is by default just P’s textual end of code.
- You can specify additional terminal states by using end-label:
 - Of the form “end_1” , “end_blabla” etc
- This does **not** check “local” deadlock!

Expressing progress requirement

- We can mark some states as progress states
 - Using “progress*” labels
- Any *infinite execution* must pass through at least one progress label infinitely many often; else violation.
- We can ask SPIN (with an option) to verify no such violation exists (non-progress cycles option).

(i)SPIN Architecture



Command line mode

```
#define N 4
byte x[N];
proctype P(byte i) {
    do
        :: x[i-1] != x[i] -> x[i] = x[i-1];
    od
}

init { run P(1); run P(2) }

ltl p0 { <>(x[0]=x[2]) }
```

- *Simulate:* spin model.pml
- *Intersect:*
spin -a model.pml
- *Compile:* cc pan.c
- *Verify:* a.exe
- With fairness: a.exe -f
- *Debug:*
spin -t -c model.pml
- Various options: see
<http://spinroot.com/spin/Man/Pan.html>

File.. Edit.. View.. Run.. Help

SPIN DESIGN VERIFICATION

Line#: 1

Find:

```
:: request?REQ2 -> { (resource == 0) ; resource=2 ; granted2!GRANTED }
od
}
```

```
active proctype customer10 {
do
:: { request!REQ1
; granted1?GRANTED
; skip /* representing customer1 uses the resource */
; resource = 0 /* after some time, freeing the resource again */
}
od
}
```

```
active proctype customer20 {
do
:: { request!REQ2
; granted2?GRANTED
; skip
; resource = 0
}
od
}
```

+ <starting simulation>
c:/apps/spin/spin425.exe -X -p
c:/apps/spin/spin425.exe -Z pa
<done preprocess>
c:/apps/spin/spin425.exe -f "!" [

Linear Time Temporal Logic Formulae

Formula: $\square \neg p$ Load...

Operators: $\square \neg \wedge \rightarrow \text{and} \text{or} \text{no}$

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes:

- Use Load to open a file or a template.

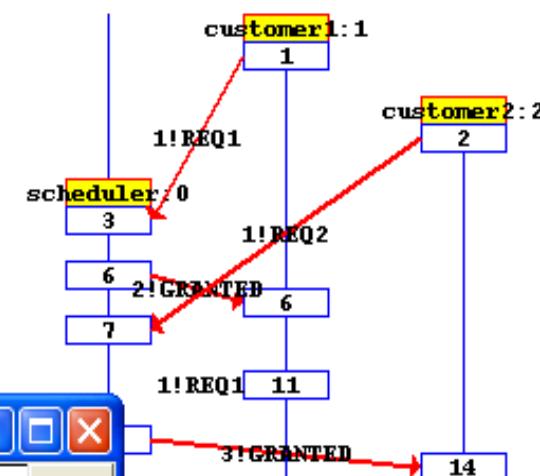
Symbol Definitions:

```
#define p (resource==1)
```

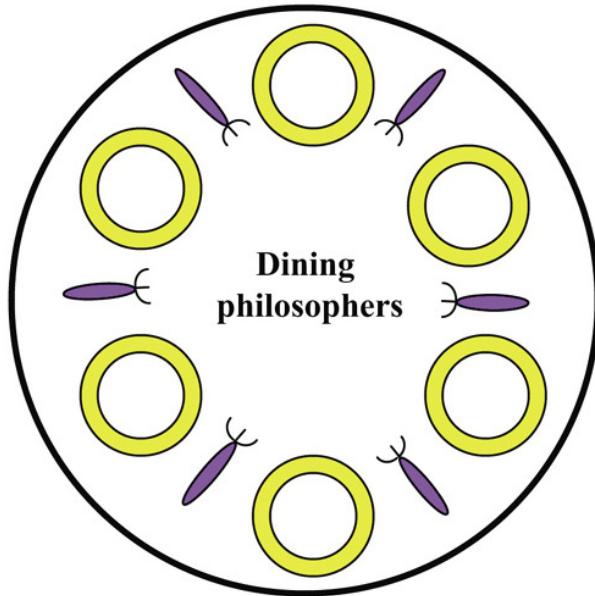
Never Claim:

```
:: (! ((p))) -> goto accept_S4
:: (1) -> goto T0_init
fi;
accept_S4:
if
:: (! ((p))) -> goto accept_S4
fi;
}
```

Verification Result: Run Verification



Dining philosophers



- N “philosopher”-processes
- Each process:
 1. acquire left and right forks (not atomic)
 2. eat...
 3. release forks
 4. think..... then go back to 1

Modelling a solution in Promela

```
#define N 4
byte fork[N] ;
bool eating[N] ;

proctype P(byte i) {
    do
        :: atomic{
            (fork[i] == N && fork[(i + 1) % N] == N) ;
            fork[i] = i
            fork[(i + 1) % N] = i } ;
            eating[i] = 1 ;// eat ...
            eating[i] = 0 ;
            fork[i] = N ;
            fork [(i + 1) % N]= N
    od
}
```

this atomicity may be unrealistic

*Deadlock free, but not starvation free
(e.g. 1 and 3 can alternatingly keep getting forks, thus starve 0)*

Creating processes and init { ... }

```
init {
    byte i ;
    ... // initialize forks
    i = 0 ;
    do
        :: i<N    -> { run P(i) ; i++ ; }
        :: i>=N   -> break ;
    od
}
```

Put this in
atomic { ... } ;
Be aware of
what it means!

How to express the specification?

```
prototype P(byte i) {  
    do  
    :: { atomic { (fork[i] == N && fork[(i + 1) % N] == N) ;  
                fork[i] = i ;  
                fork[(i + 1) % N] = i } ;  
  
        eating[i] = 1 ;  
        eating[i] = 0 ;  
        fork[i] = N ;  
        fork [(i + 1) % N]= N  
    }  
    od  
}
```



Does not exclude the situation that after the check $P(i)$ lost a fork!

```
assert (fork[i] == i &&  
       fork[(i+1)%N]== i)
```

Better option, with ltl:

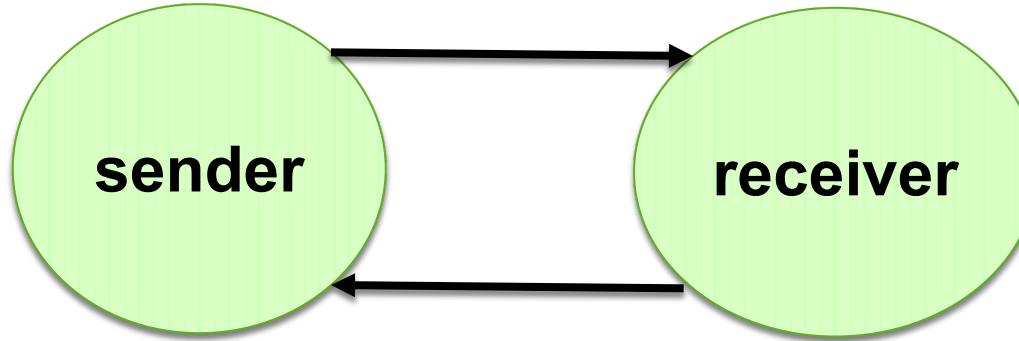
$\Box(eating[0] \rightarrow fork[0] == 0 \ \&\ fork[1]== 0)$
... similarly for other processes

Using a “monitor” process

```
active proctype monitor() {  
    assert(  
        (!eating[0] || (fork[0]==0 && fork[1%N]==0))  
        && (!eating[1] || (fork[1]==1 && fork[2%N]==1))  
        && ...  
    ) ;  
}
```

*But we still can't express that if a process is “hungry”, it will eventually eat.
Using progress labels does not seem sufficient either. For more general
temporal specification, use of LTL formulas.*

Example: Alternating bit protocol



- imperfect “connections”, but corrupted data can be detected (e.g. with checksum etc).
- Possible solution: send data, wait for a positive acknowledgement before sending the next one.
Just 1 bit is needed for the ack, hence the “bit” in the name.

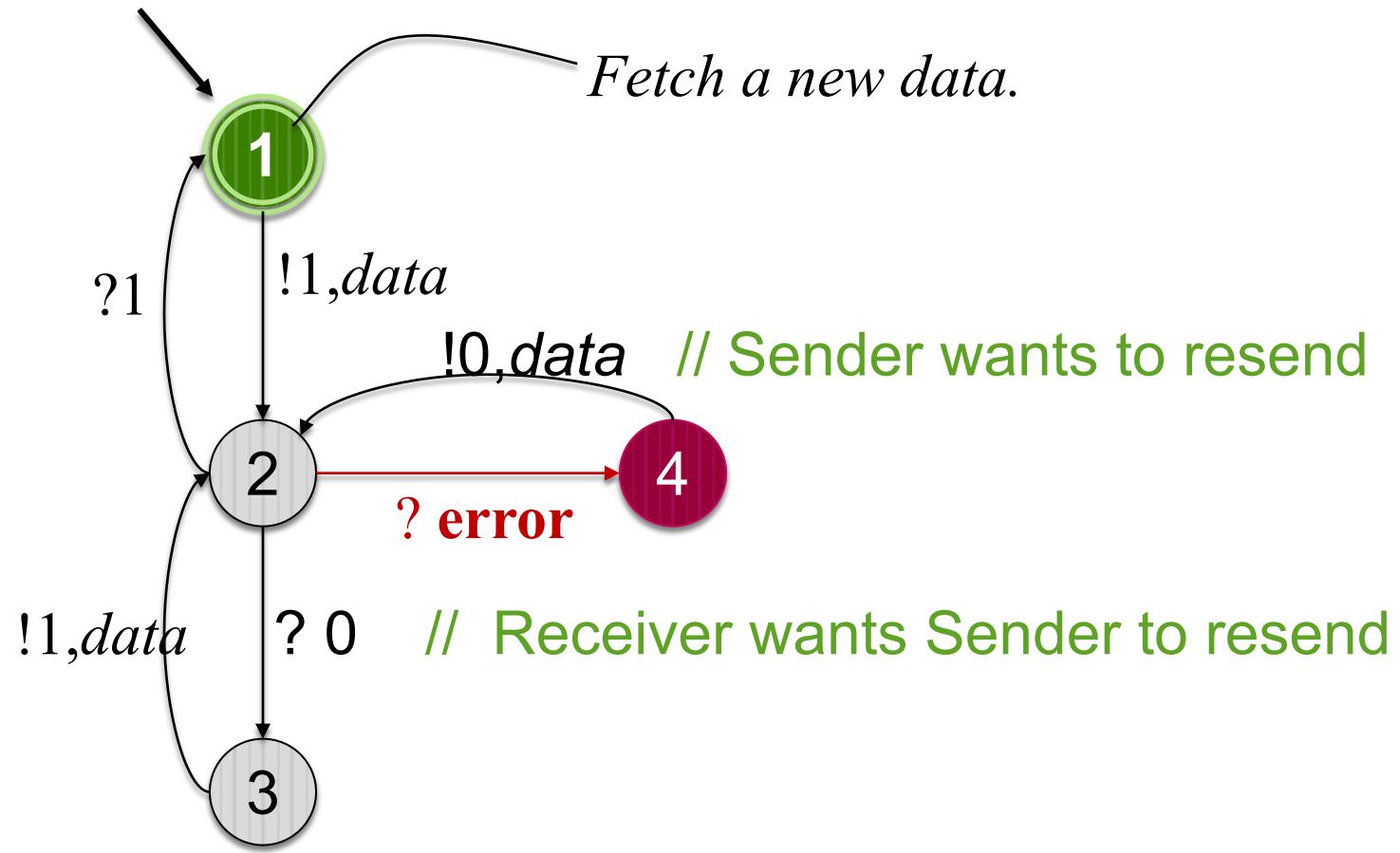
You can think of several ways to work it out...

- *A note on reliable full-duplex transmission over half-duplex links*, K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, Communications of the ACM, Vol 12, 1969.
 - NPL Protocol
 - M<2 Protocol (we'll discuss this one)
- For more, check out:

<http://spinroot.com/spin/Man/Exercises.html>

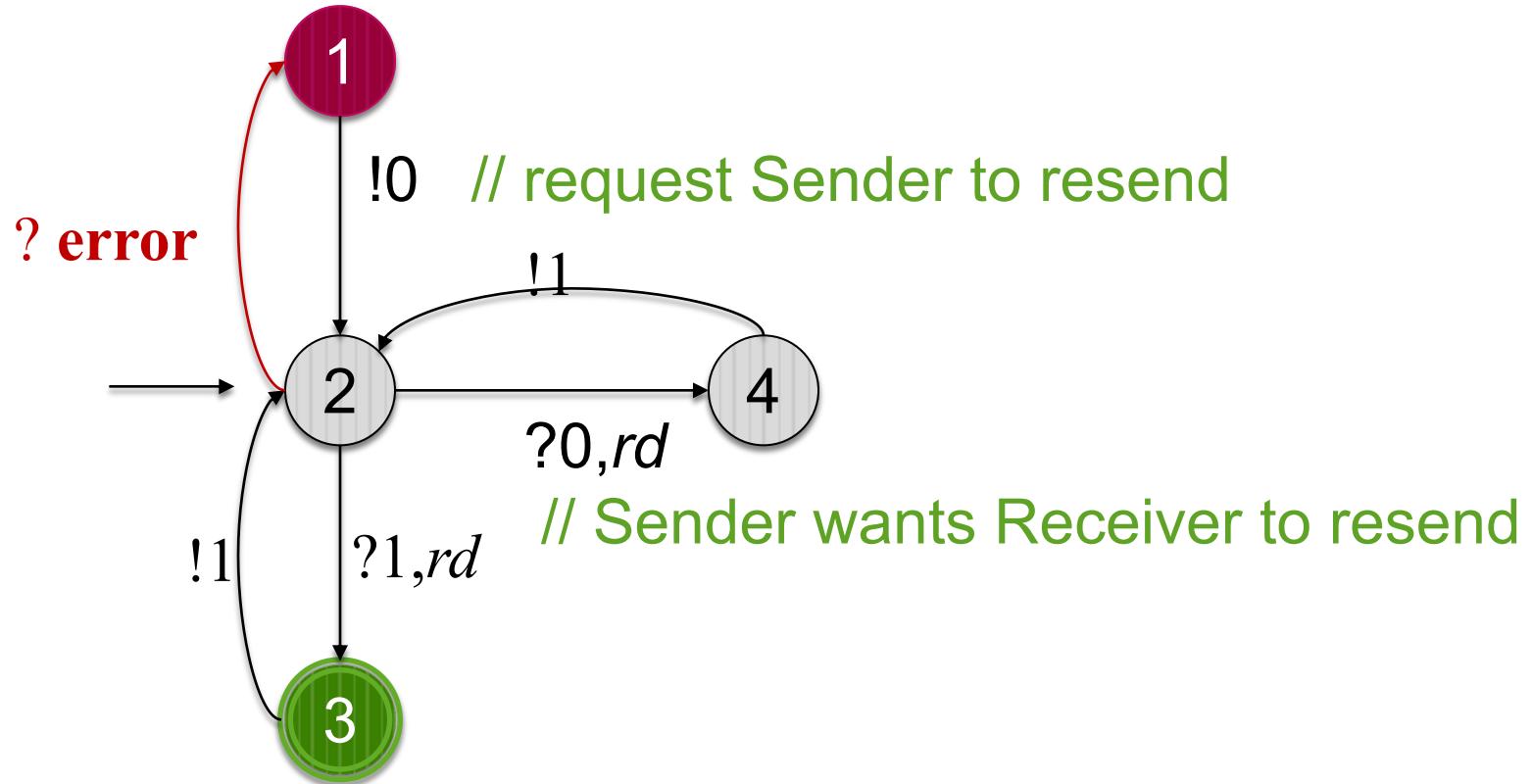
e.g. Go-Back-N Sliding Window Protocol

M<2 Protocol, Sender part

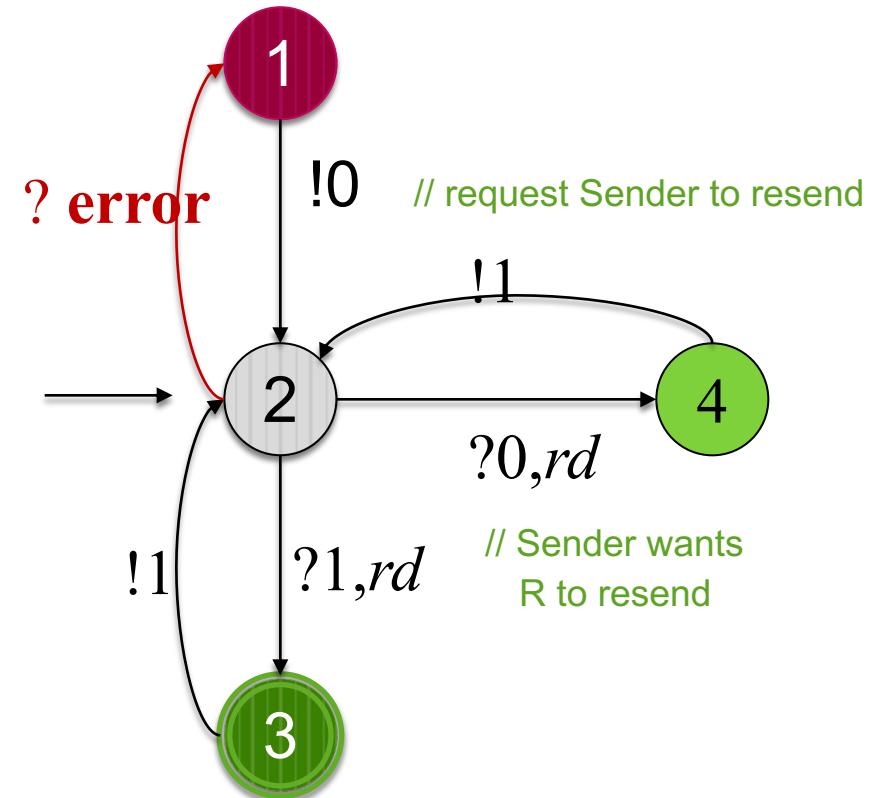
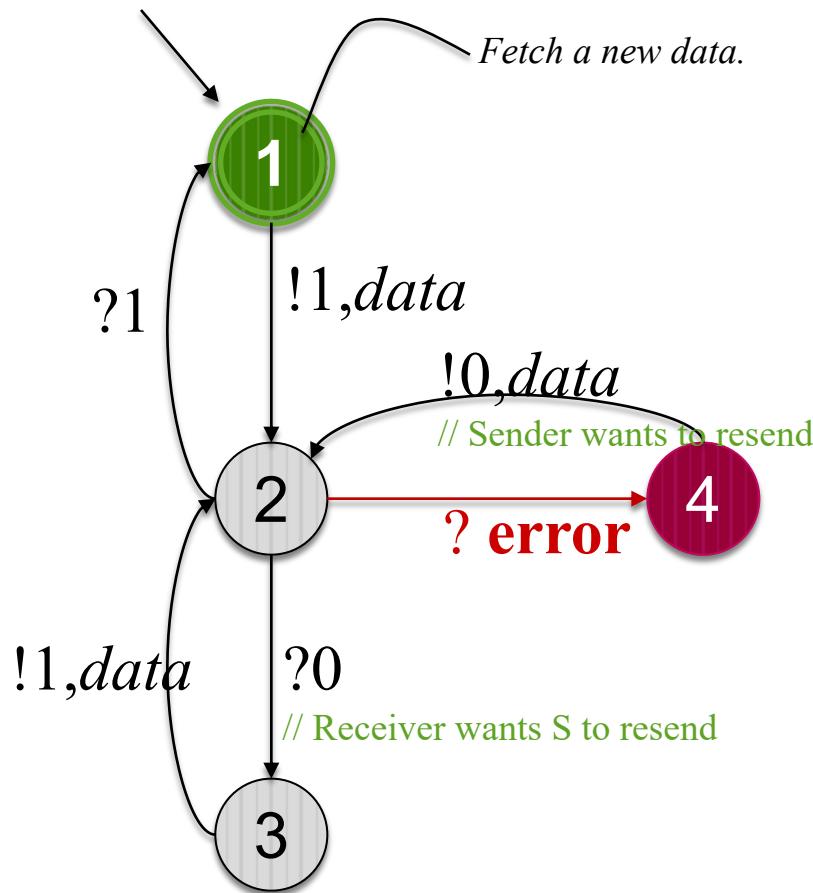


State 1 is the **starting state**, and its **accepting state** in the sense when the sender is in this state, it assumes the last data package it sent has been successfully received by the receiver, and so it fetches a new data package to send.

M<2 Protocol, Receiver part



Scenario: 1x error, corrected



Though each automaton is simple, the combined (and concrete) behavior is quite complex; ≈ 100 states in my (abstract) SPIN model (there are more explicit states, if we take the “data” into count).

Modeling in Promela

```
chan S2R = [BufSize] of { bit, byte } ;
chan R2S = [BufSize] of { bit } ;

proctype Sender (chan in, out) { ... }

proctype Receiver(chan in, out) { ... }

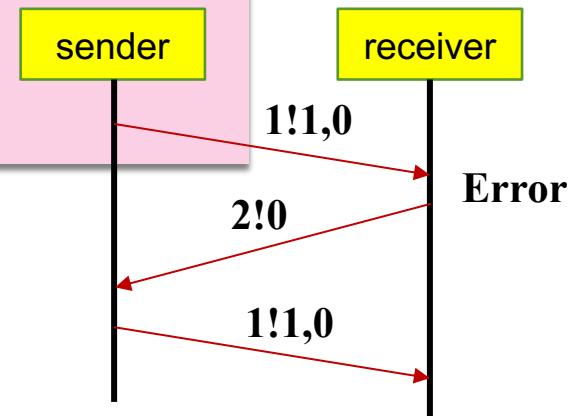
init {
    run     Sender(R2S, S2R)   ;
    run     Receiver(S2R, R2S)
}
```

Modelling in Promela

```
proctype Receiver(chan in, out) {
    show byte rd ; /* received data */
    show bit cbit ; /* control bit */
    show byte accepted ;
    do
        :: in ? cbit, rd ;
        if
            :: (cbit == 1) -> accepted=rd ; out!1
            :: (cbit == 0) -> out!1
            :: printf("MSC: ERROR\n") ; out!0
        fi
    od
}
```

So, how big the channels should be? Is 0 good enough?

Using visualization in iSpin:



A different style, with “goto”

```
proctype Sender(chan in, out) {
    show byte data ; /* message data */
    show bit cbit ; /* received control bit */

    S1: data = (data+1) % MAX ; out!1,data ; goto S2;

    S2: in ? cbit ;
        if
        :: (cbit == 1) -> goto S1
        :: (cbit == 0) -> goto S3
        :: printf("MSC: ERROR\n") -> goto S4
        fi ;

    S3: out!1,data ; goto S2 ;

    S4: out!0,data ; goto S2 ;
}
```

Specifications

The protocol does not deadlock.

- With LTL

$$\square \Diamond \text{Sender}@S2 \wedge \square \Diamond \text{Receiver}@S2$$

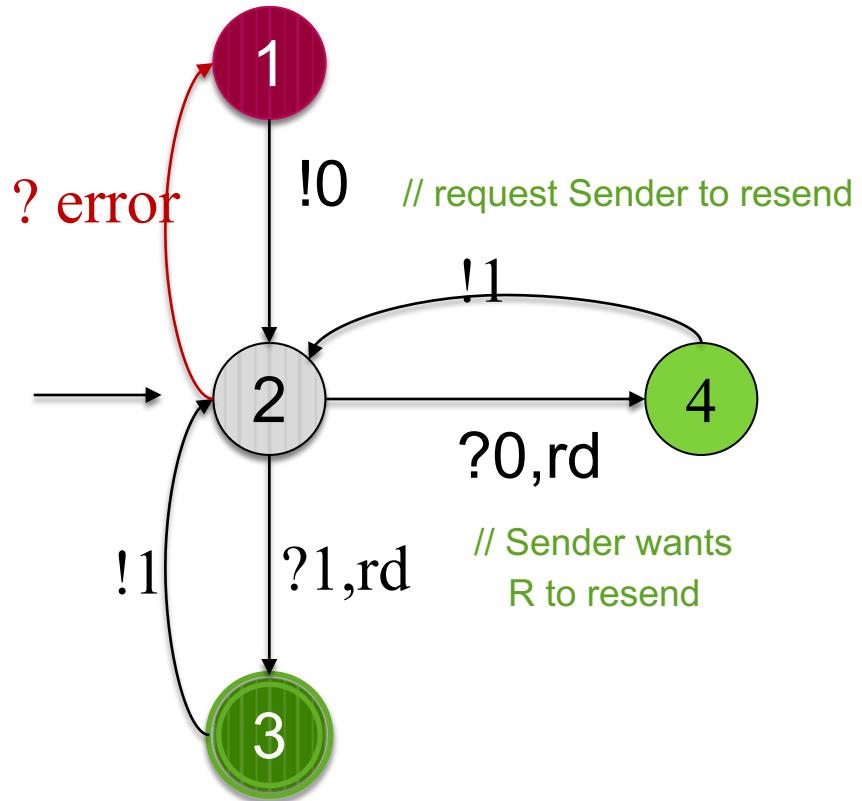
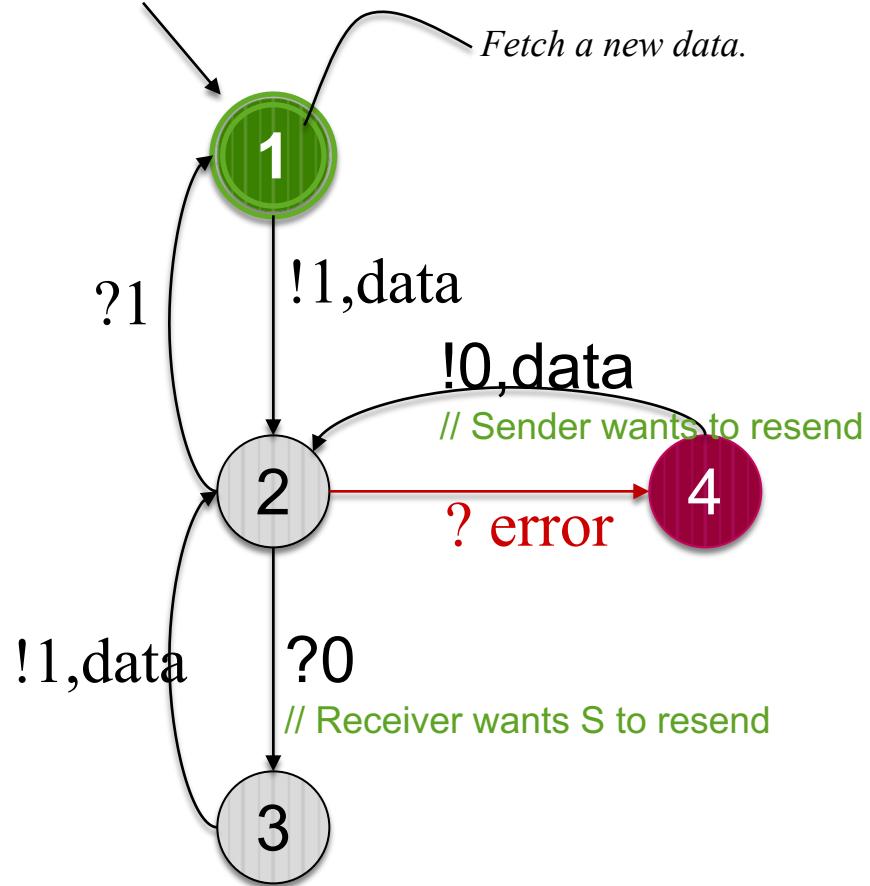
Specifications

Each data package, if accepted by the receiver, is accepted exactly once!

- LTL:

$$\Box(Receiver@S3 \rightarrow (Receiver:accepted == Sender:data))$$
$$\Box(Sender@S1 \rightarrow (Receiver:accepted == Sender:data))$$

2x successive errors



Ouch...

Ok... but suppose we still want to verify these:

But, if error does not occur twice successively then: every pck sent, if accepted, is accepted exactly once.

If no error occur, every data sent will eventually be accepted.

Rather than writing a constrained model, just extend the model to allow you to count successive errors. Use LTL of the form $\varphi \rightarrow \Phi$ to allow you quantify over executions of certain properties.