Modeling and Verification with SPIN

Wishnu Prasetya

wishnu@cs.uu.nl

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

Overview

- Architecture & a bit more about SPIN
- SPIN's modeling language
- Examples of models in SPIN

- Acknowledgement: some slides are taken and adapted from Theo Ruys's SPIN Tutorials.

Spin and Promela

- SPIN = *S*imple *P*romela *In*terpreter

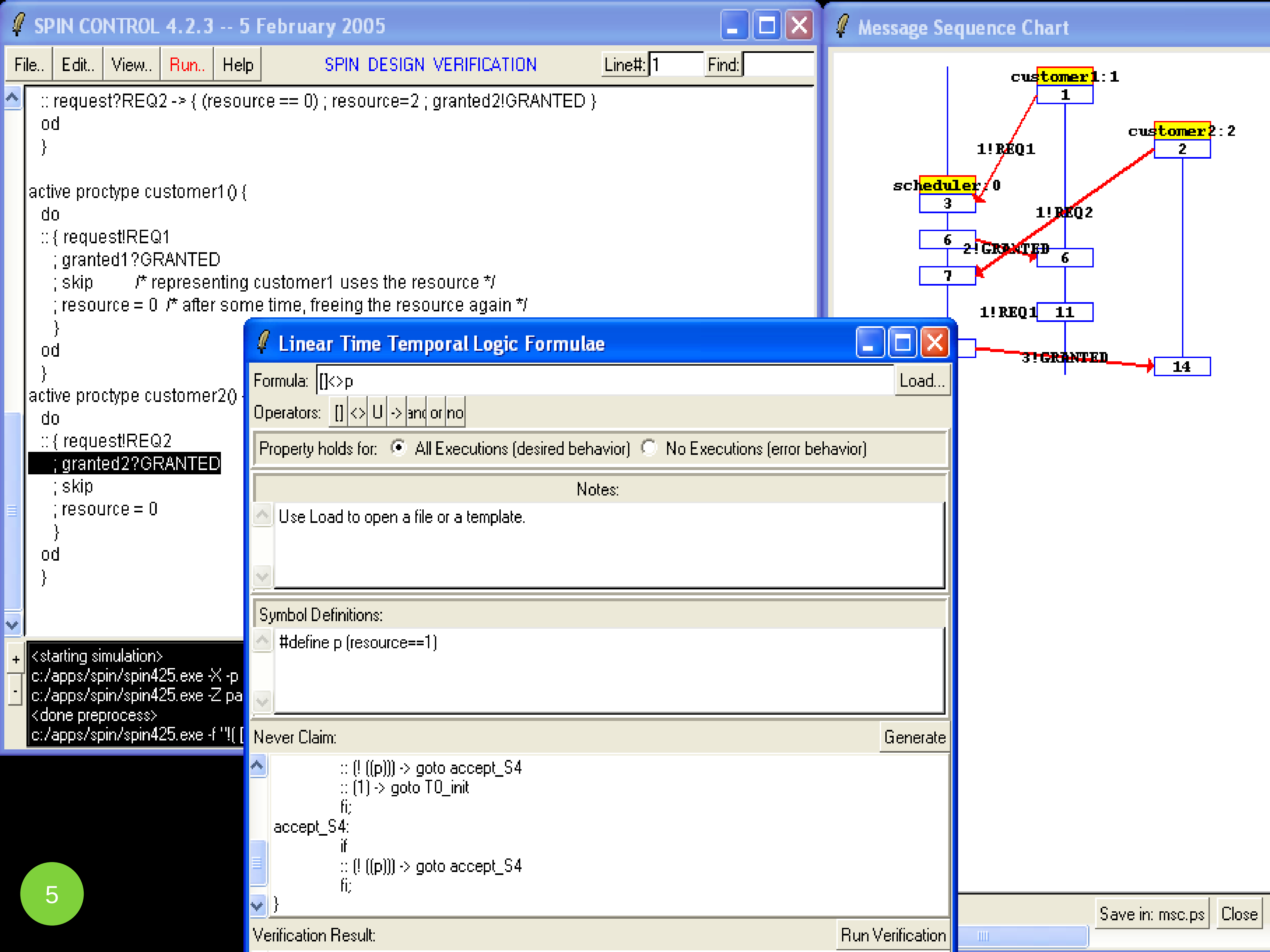
- Promela = *Pro*cess *Me*ta *La*nguage
  - Is a *modelling* language! (not a language to build an application)

- Strong features :
  - Powerful constructs to synchronize concurrent processes
  - Cutting edge model checking technology
  - Simulation to support analysis (of the models)

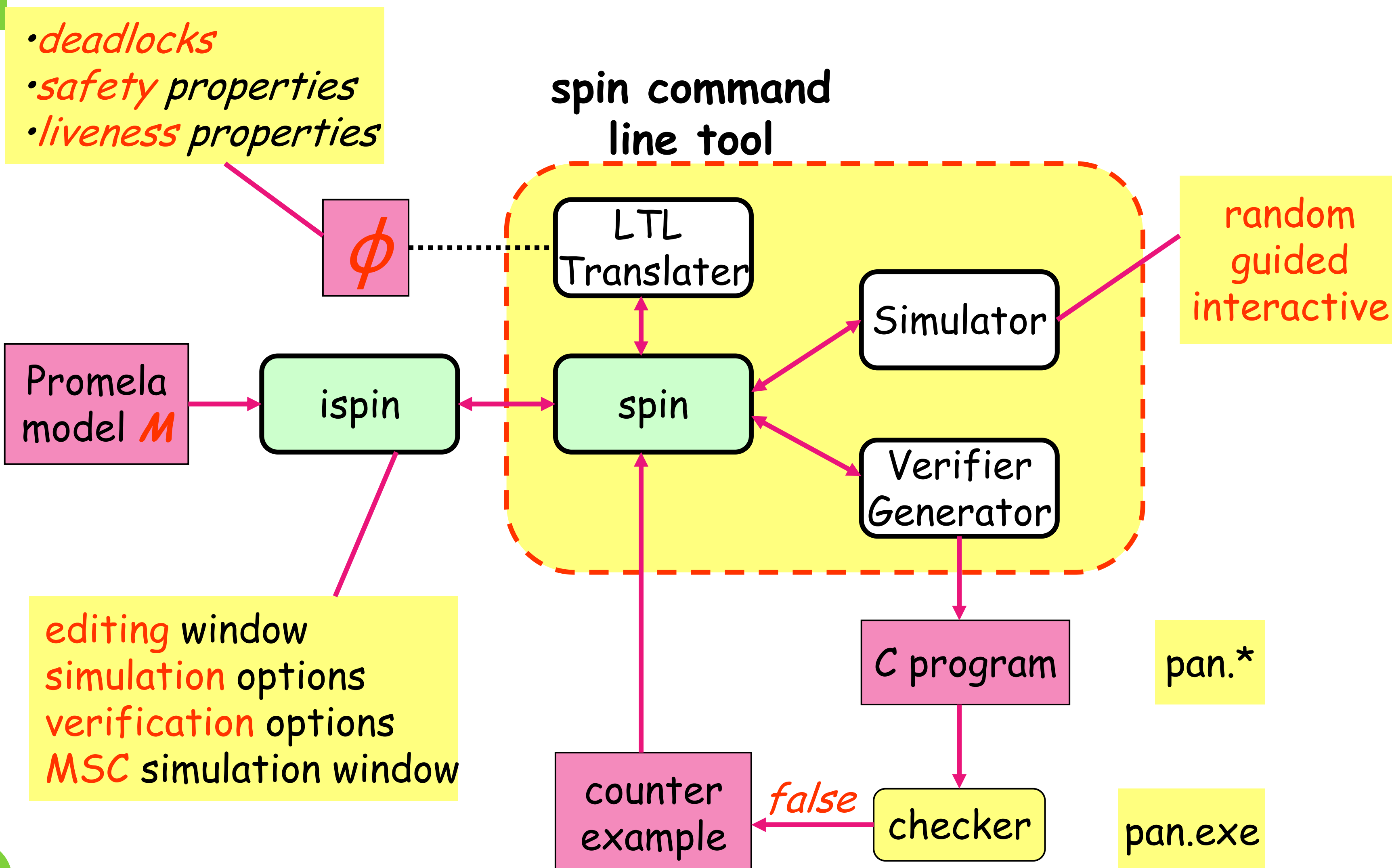
SPIN

- Concurrency is a hot area again, now that we all use multi-core CPUs.

- Other applications:
  - *AnWeb: a system for automatic support to web application verification*, Di Sciascio et al, in 14th conf. on Soft. Eng. and knowledge eng., 2002.
  - *Privacy and Contextual Integrity: Framework and Applications*, Barth et al, in IEEE Symposium on Security and Privacy, 2006.


SPIN CONTROL 4.2.3 -- 5 February 2005
File.. Edit.. View.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 1 Find:
:: request?REQ2 -> { (resource == 0) ; resource=2 ; granted2!GRANTED }
od
}
active proctype customer1() {
do
:: { request!REQ1
; granted1?GRANTED
; skip /* representing customer1 uses the resource */
; resource = 0 /* after some time, freeing the resource again */
}
od
}
active proctype customer2()
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 "!([
Message Sequence Chart
customer1:1
1
customer2:2
2
1!REQ1
scheduler:0
3
1!REQ2
6
2!GRANTED
6
7
1!REQ1
11
3!GRANTED
14
Save in: msc.ps
Close
Linear Time Temporal Logic Formulae
Formula: []<>p
Load...
Operators: [] <> U -> and or not
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:
Generate
:: (! ((p))) -> goto accept_S4
:: (1) -> goto T0_init
fi;
accept_S4:
if
:: (! ((p))) -> goto accept_S4
fi;
}
Verification Result:
Run Verification

(X)SPIN Architecture

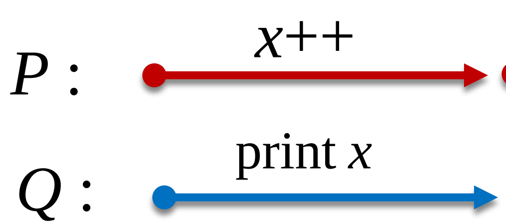


System, process, and action.

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

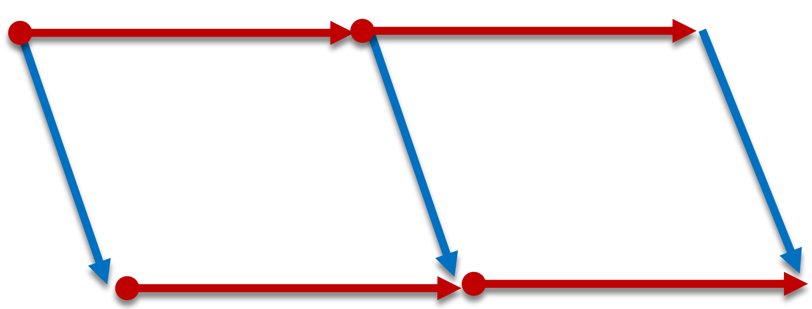
Interleaving model of concurrency

- Consider (with pseudo notation):



Assume each arrow is *atomic.*

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



(note the interleaving)

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
  - $0 \in S \rightarrow \mathit{found}:=\mathrm{true}$ ....?

Example

```
byte  x = 1 ;

active proctype  P1() { x++ ;  assert (x==2)  ; }

active proctype P2() { x-- ; }
```

(using a global variable to interact)

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 macro
- 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

Multiprogramming 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.

loop : do-statement

```
do
:: stmt_1
:: …
:: stmt_n
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.

Non-determinism can be useful for modeling

```
active proctype consumer() {
   do

   ::  c ? x ;

   ::  c ? x ; x=corrupted ;  //  to model occasional corrupted data

  od
}
```

Exiting a loop

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

```
do
   :: (i>0)   → i--
   :: (i==0) → break
do
```

```
do
   :: { i-- ; (i>0) }
   :: break
do
```

Label and jump

```
L0:  (x==0)  ;
if
::  …  goto L0 ;
::  …
fi
```

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

  <> 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.

Expressing local 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)

But we can also express global invariant!

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

```
byte x=0 , y=0

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

active proctype Q { (x>0) ; y++ ; (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

Deadlock checking

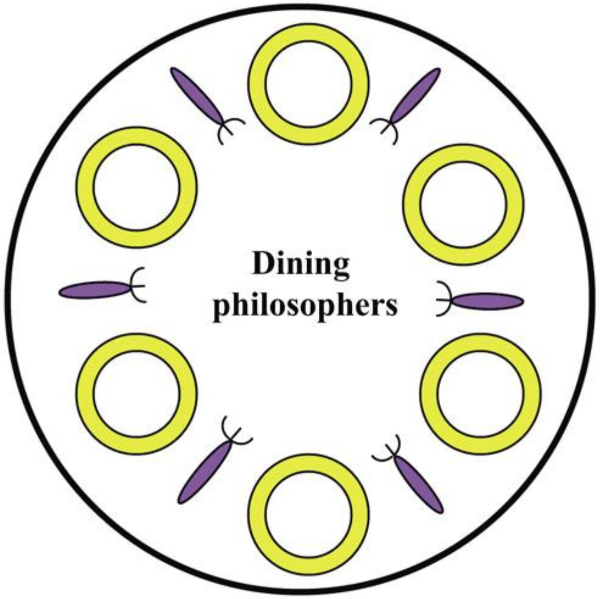
- When a system comes to a state where it has no enabled transition, but one of its processes is not in its terminal (end) state:
  - Deadlocked, will be reported by SPIN
  - But sometimes you want to model that this is ok → suppress it via the invalid-endstate option.

- 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

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).

Dining philosophers



- N philosophers
- Each process:
  1. grab left and right fork simultaneously
  2. eat...
  3. release forks
  4. think................ then go back to 1

The processes in Promela

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

proctype P(byte i) {
   do
   :: (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
}
```

atomic { ..... }

- *Why use bytes ?*
- *Should we enable the default end-state checking?*
- *How to instantiate the P(i)'s ?*
- *Ehm... this is not correct !*

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!

What if we want to show that the algorithm is still correct for any initial value of forks, as long as you have at least one pair of forks free at the beginning, and hat forks are only taken in pairs?

Using non-determinism to quantify over your data

```
init {
  // initializing the array x
  byte i = 0 ;  byte v ;
  do
  :: i>=N -> break ;
  :: { if
       :: v = N
       :: v = i
       fi ;
       fork[i]=v ; fork[(i+1)%N]=v ;
       i++ ;
     }
  od ;
   …. // now create the processes as in the previous
slide
}
```

How to express the specification?

```
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 ;
         eating[i] = 0 ;
         fork[i] = N ;
         fork [(i + 1) % N]= N
       }
  od
}
```

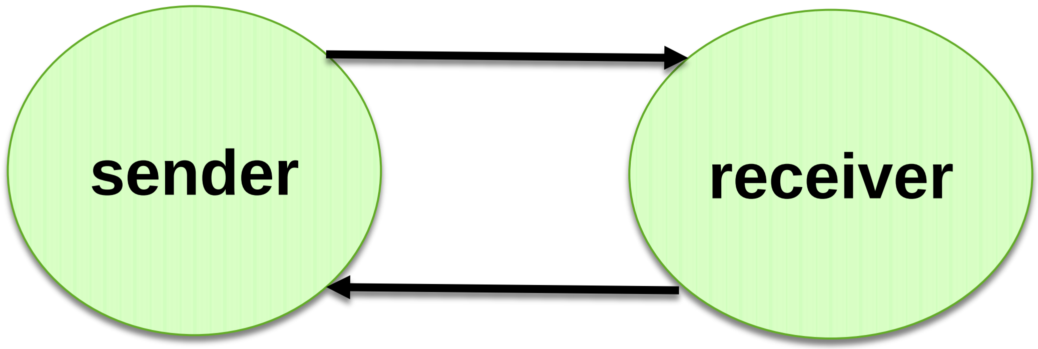
assert (fork[i] == i &&
fork[(i+1)%N]== i)

Using a “monitor” process

```
active proctype monitor() {
   byte i ;
   i = 0 ;
   do
   :: i>=N -> break ;
   :: i<N  -> {
        assert(!eating[i]
               ||
               (fork[i]==i && fork[i+1%N]==i)) ;
        i++ ;
      }
   od
}
```

But we still can’t express that if a process is “hungry”, it will eventually eat. In this particular problem, we can still express it using progress labels. For more general temporal specification, we will look at the 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.

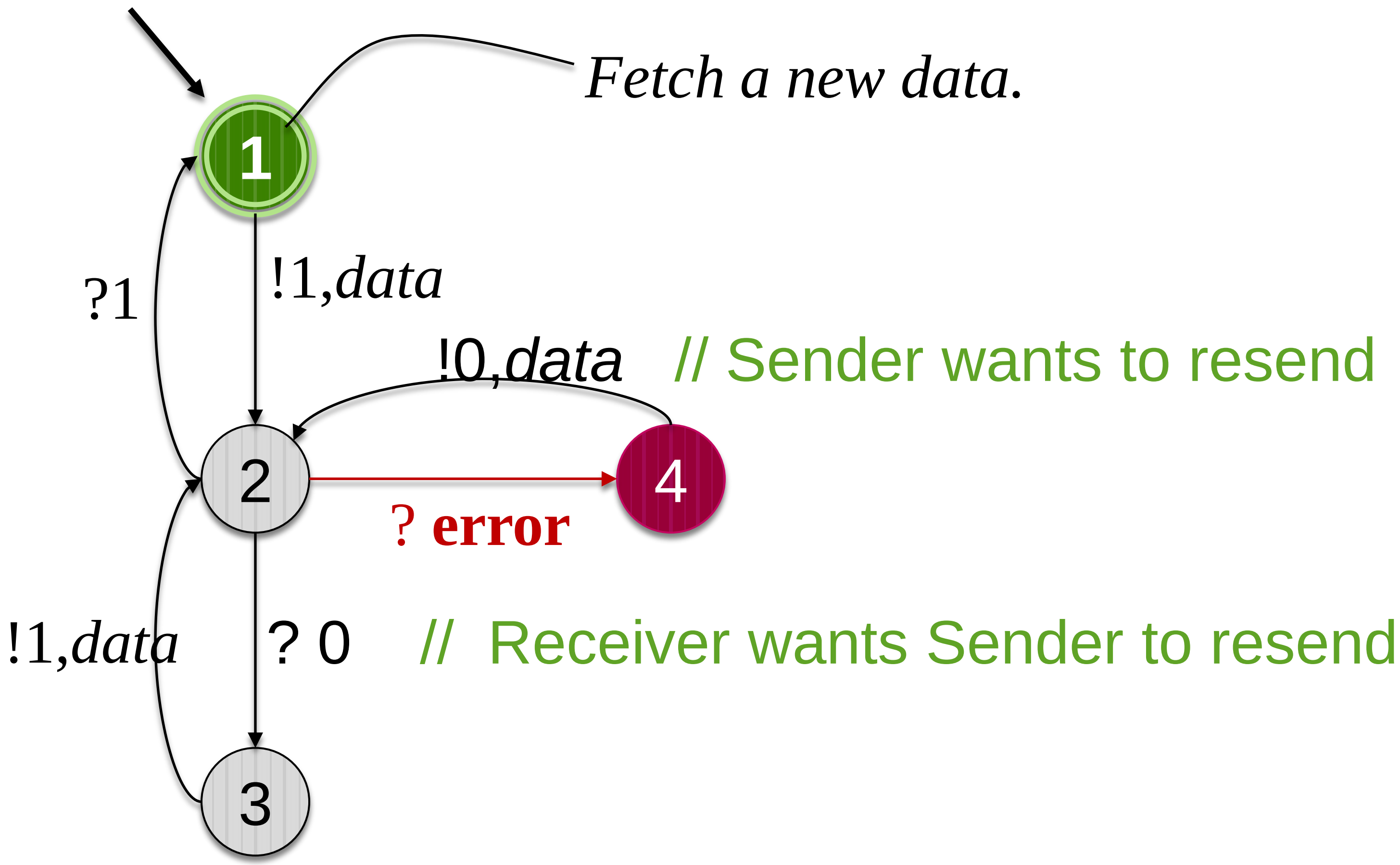
You can think of several ways to work it out...

- *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

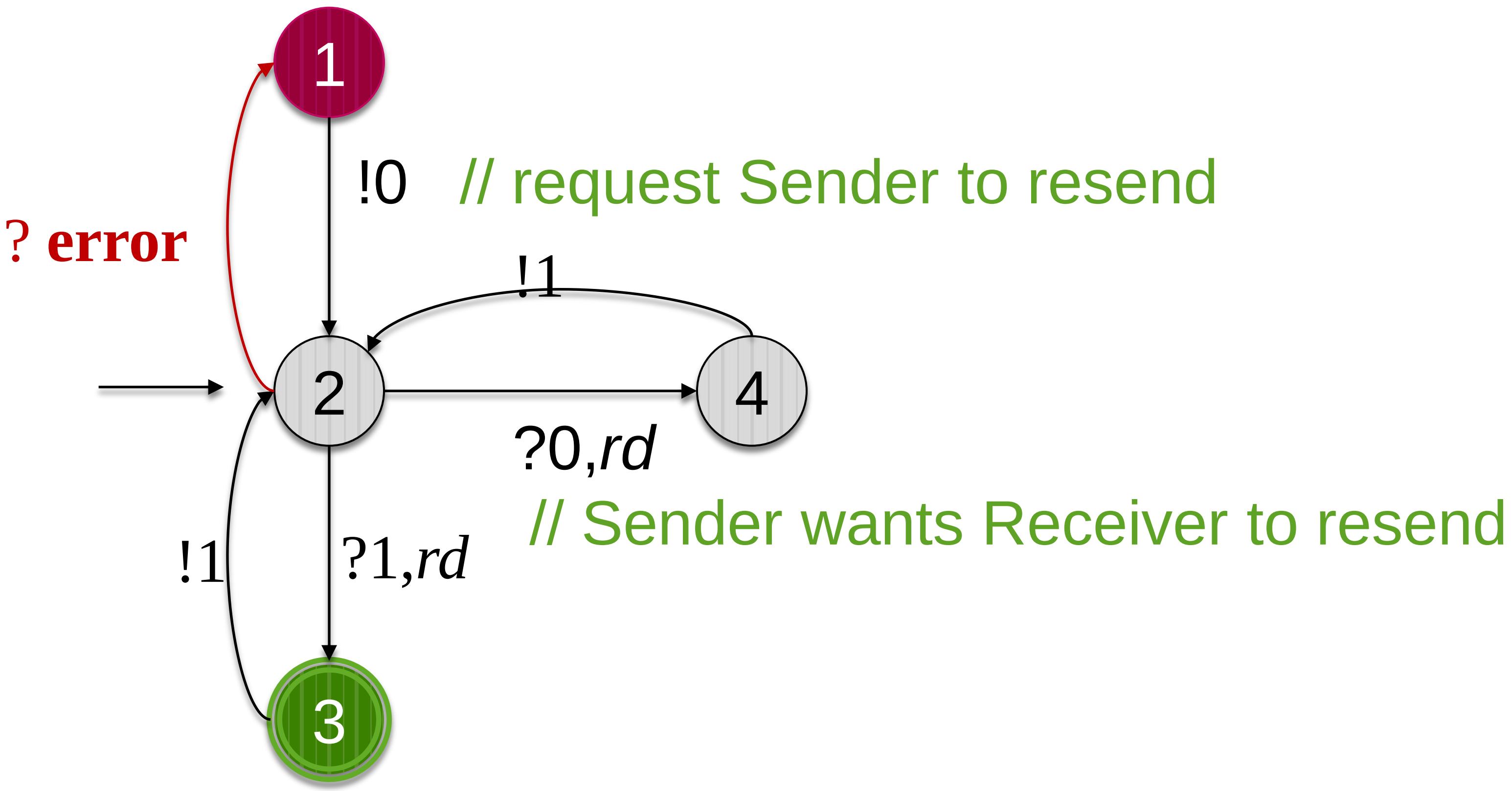
  e.g. Go-Back-N Sliding Window Protocol

M<2 Protocol, Sender part

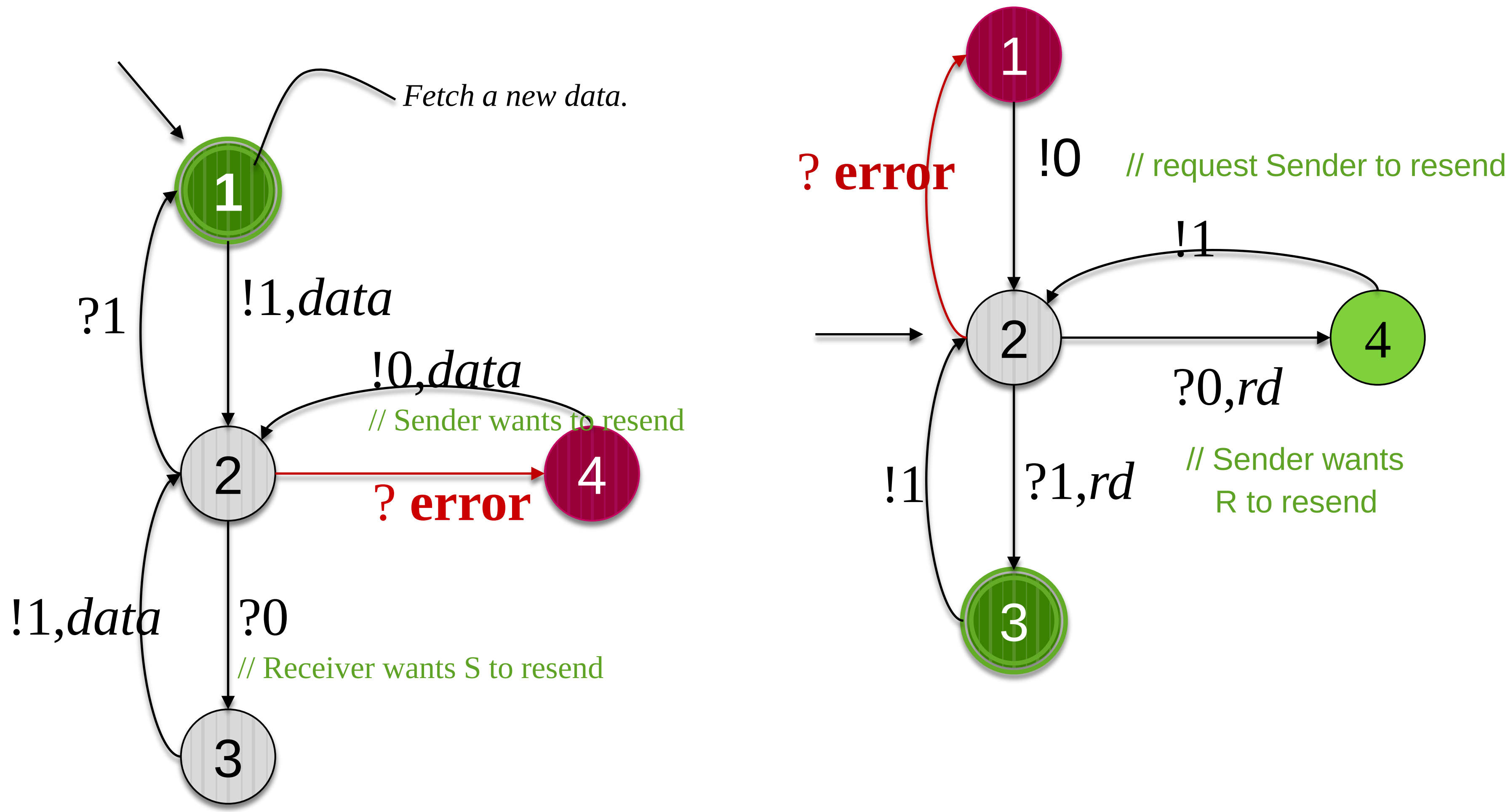


State 1 is the **starting state**, and its **accepting state** in the sense when the sender is in this state, it assumes the last data package it sent has been successfully received by the receiver, and so it fetches a new data package to send.

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).

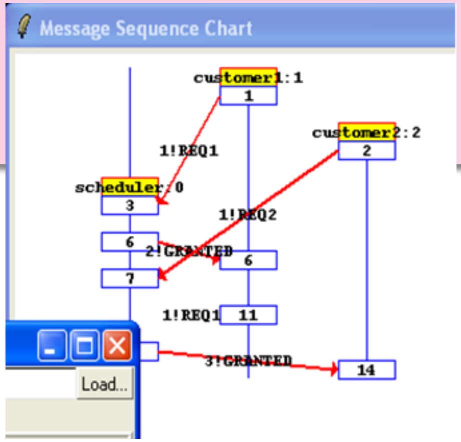
Modeling in Promela

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

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

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

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

Modelling in SPIN

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

    do
    :: in ? cbit, rd ;
       if
       :: (cbit == 1) -> out!1
       :: (cbit == 0) -> out!1
       :: printf("MSC: ERROR1\n") ; out!0
       fi
    od
}
```

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



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: AERROR1\n") -> goto S4
        fi ;

    S3: out!1,data ; goto S2 ;

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

Specification, with assertions?

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

- This time, not possible with assertions (at least not without the help of 'something else').
- In LTL (to be discussed later), we can try something along this line :

$$\Box(Receiver@S3 \rightarrow (Receiver@rd == Sender@data))$$

- But this still does not quite express the above.

Specification, using shadow variables

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

- Extend the model with 'shadow variables'
  - Are used purely for expressing specifications
  - Must not influence the original behavior
- In our case:
  - exploit that sender generates new data by data+1
  - introduce a shadow variable "*last*" → previously accepted data
  - Impose this assertion on the acceptance state (of Receiver):

current data to be accepted = *last* + 1

Extending the model

```
proctype Receiver(chan in, out) {
    show byte rd    ;  /* received data */
    show bit  cbit   ;  /* contro

    do
    :: in?cbit,rd ;
       progress:
       if
       :: (cbit == 1) -> out!1
       :: (cbit == 0) -> out!1
       :: printf("MSC: ERROR1\n") ; out!0
       fi
    od
}
```

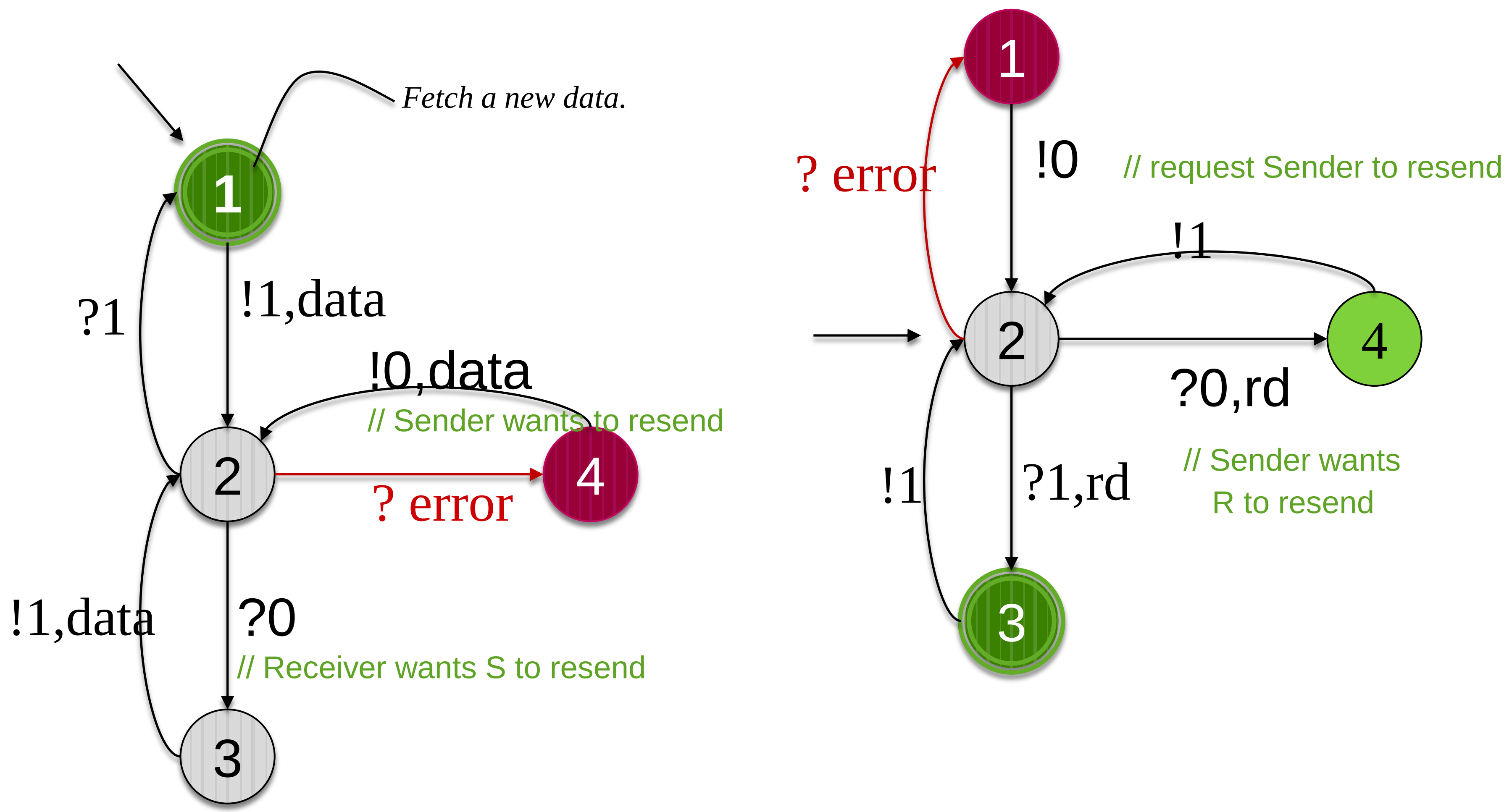
show byte last

assert (rd == (last+1) % MAX) ;

last = rd ;

This is the Receiver's accepting state S3

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.

The first can be expressed simply by constraining the model, namely how it simulates error.

The 2^{nd} one can't be expressed with just assertions and shadow variables.

Alternative: *LTL*.

More on Promela

Exception/Escape

- S 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 becomes executable if there is no other process the system is executable/enabled
  - so, it models a global timeout
  - useful as a mechanism to avoid deadlock
  - beware of statements that are always executable.