PROMELA/SPIN TUTORIAL

Spin [Holzmann 1991] is one of the most powerful model checkers

- has been around for a long time with many improvements by several contributors
- proven use through application in many real contexts

2001 ACM Software System Award

– joining the likes of: Unix, TCP/IP, Eclipse, Java, Tcl/Tk, Apache, TeX, PostScript, …

Spin success factors

- Push-button *on-the-fly* verification
  - *Verification performed while exploring the state space*
- Very efficient implementation in C
- Compatibility with C (accepts inlined C code)
- Availability of GUIs (iSpin, JSpin, SpinRCP)
- Support for Linear Temporal Logic (LTL) properties
- Many high-impact, real-world applications (not just a research tool)
- Embodies more than three decades of research
- Use of many advanced optimization algorithms
- Well supported and subject of active research
  - Annual conference since 1995

Class Examples and Exercises

https://github.com/cmusv-svvt/spin-public

README.md specifies Promela Coding Style

Start Promela Exercises

Create a new Promela project called “Exercises”

- SpinRCP:
  - Right-click in Model Navigator (left pane)
  - *New -> Project* (*Show wizards*) -> *General -> Project*
    - *Project Name: Exercises -> Finish*

Source: spin-public/Mutex

Mutex example in Promela

```
while (true) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  x = 0;
}
```

global variables

```
byte x = 0;
byte y = 0;
int mutex = 0;
```

```
while (true) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  y = 0;
}
```

Create a new model Mutex.pml
Run SpinRCP with…

- Syntax Check — Mutex property ok
- Verification >Safety (state properties) ➔ Passes
  - Assertion violations
  - Invalid end states
- Verification > Liveness (cycles/seq) ➔ **Fails** — Progress property not ok
  - Acceptance cycles
  - Apply never claim
  - In-model LTL formula name: progress_p
-

```
proctype X() {
      do
      ::    x = 1;
            do
            :: (y == 1)
            :: (y != 1) -> break;
            od;
            mutex++;
            // critical section
            // access r
            mutex--;
            x = 0;
      od;
}
```

process X

```
proctype mutex_p() { assert(mutex < 2); } // mutex property

init {
  run X(); run Y();
  run mutex_p();
}

ltl progress_p { <>(mutex > 0) } // progress property
```

properties and init process

```
proctype Y() {
      do
      ::    y = 1;
            do
            :: (x == 1)
            :: (x != 1) -> break;
            od;
            mutex++;
            // critical section
            // access r
            mutex--;
            y = 0;
      od;
}
```

process Y

Spin = Simple Promela Interpreter is the tool

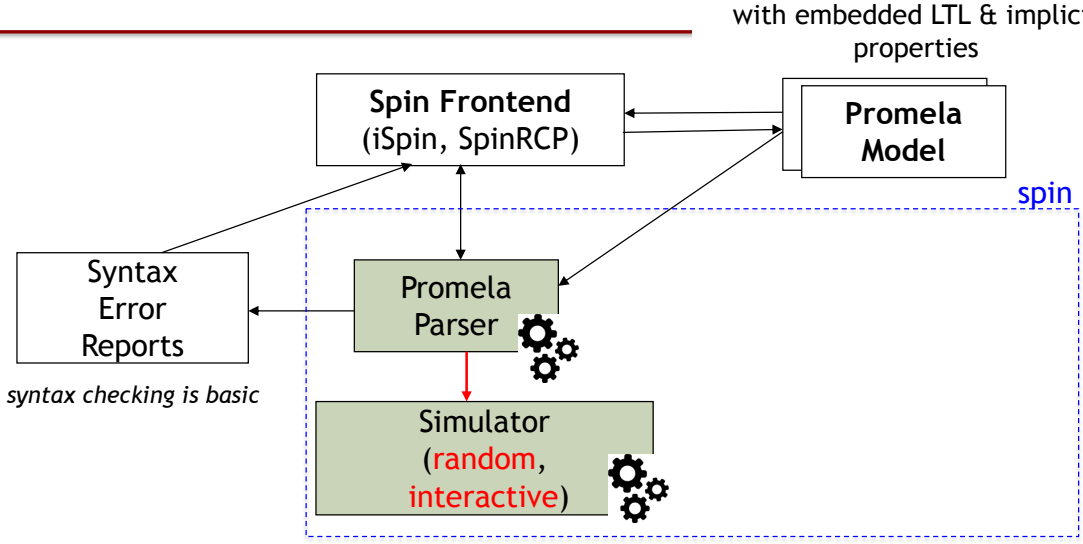
Promela = PROtocol META Language is the modeling language

with a “C-like” syntax

Does not require prior knowledge of C beyond general knowledge of how programming languages work! Promela is an entirely different kind of language with a completely different execution semantics than other programming languages you have seen!

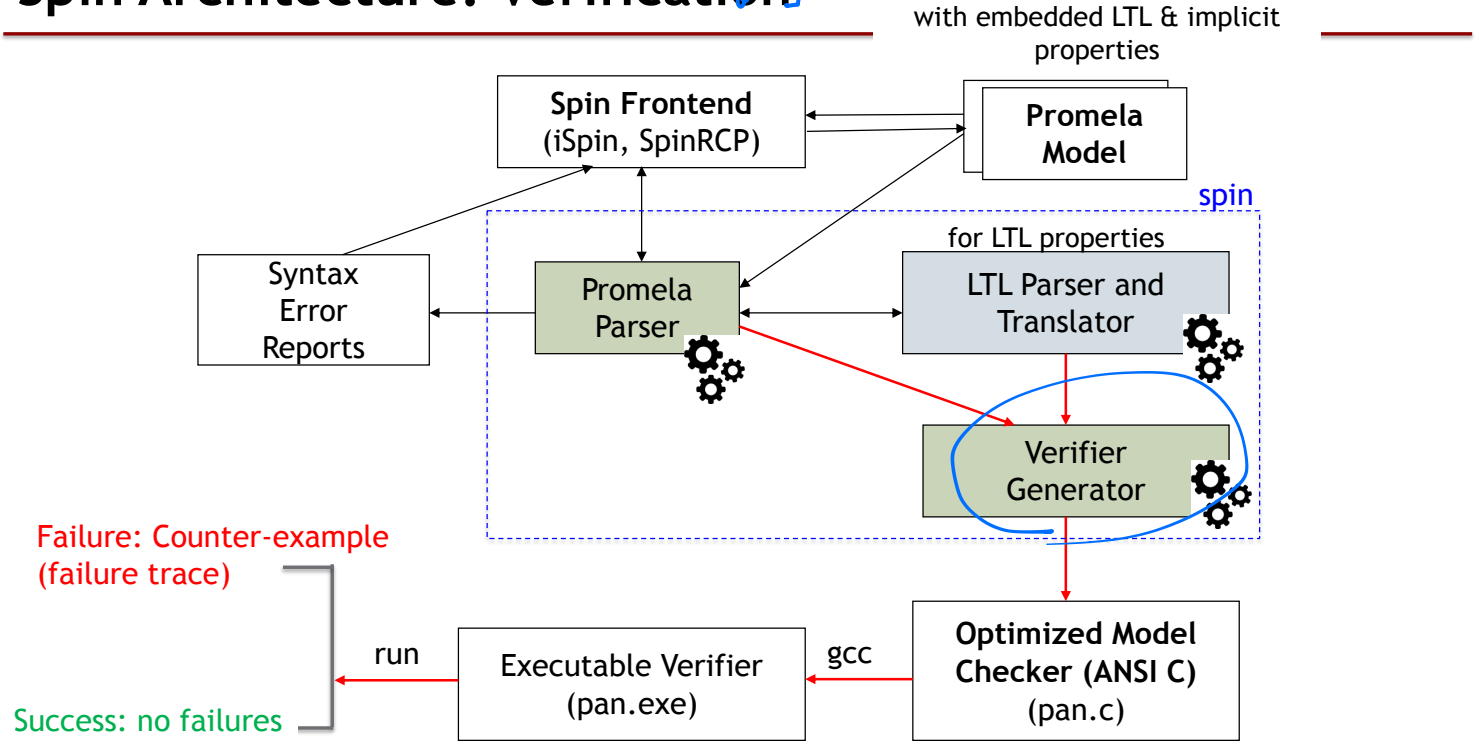
```
mtype = { MSG, ACK }
chan toS = [1] of { mtype }
chan toR = [1] of { mtype }
bool flag = false;

proctype Sender() {
   toR!MSG;
   toS?ACK;
   flag = true;
}

proctype Receiver() {
   toR?MSG;
   toS!ACK;
}

init {
   run Sender();
   run Receiver();
   timeout;
   assert(flag);
}
```

Spin Architecture: Simulation

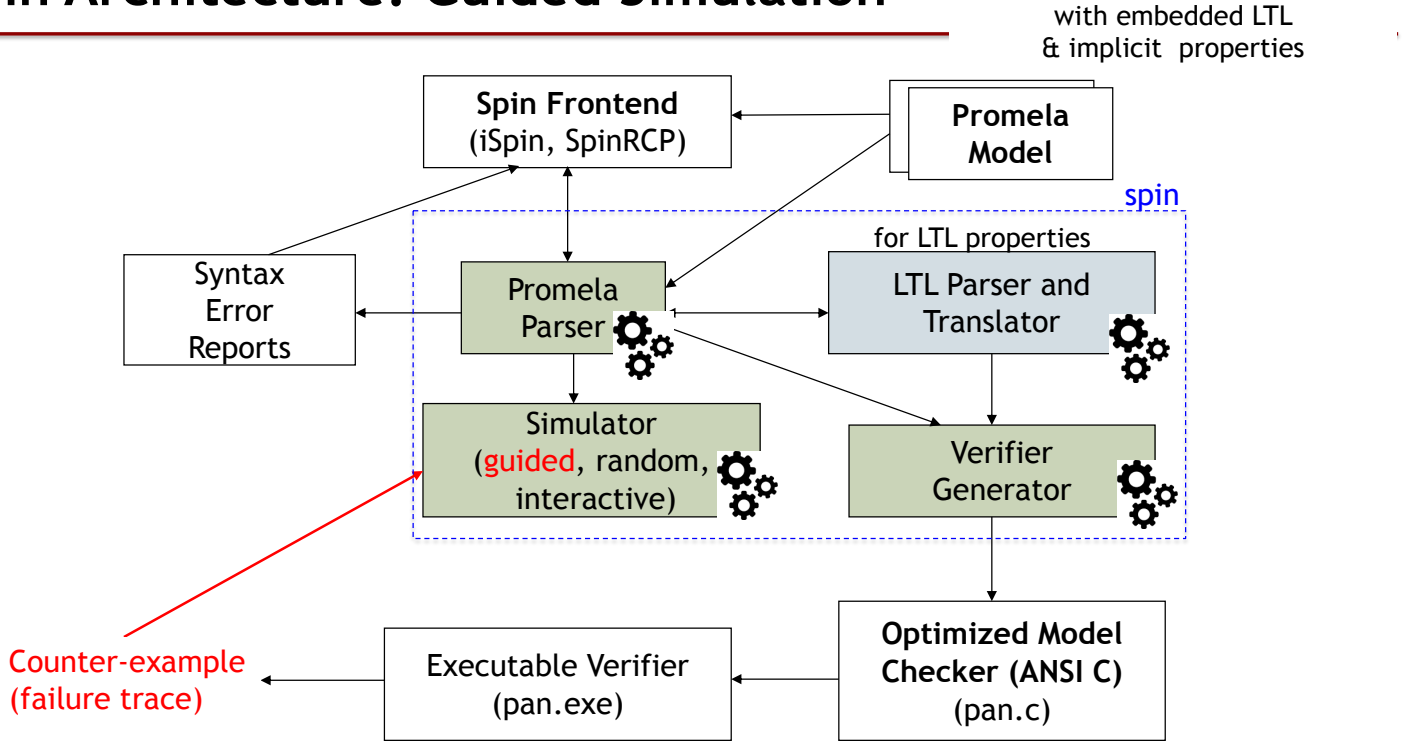


random: execute model by picking a random path
interactive: allow user to choose the path every time more than one execution choice is possible

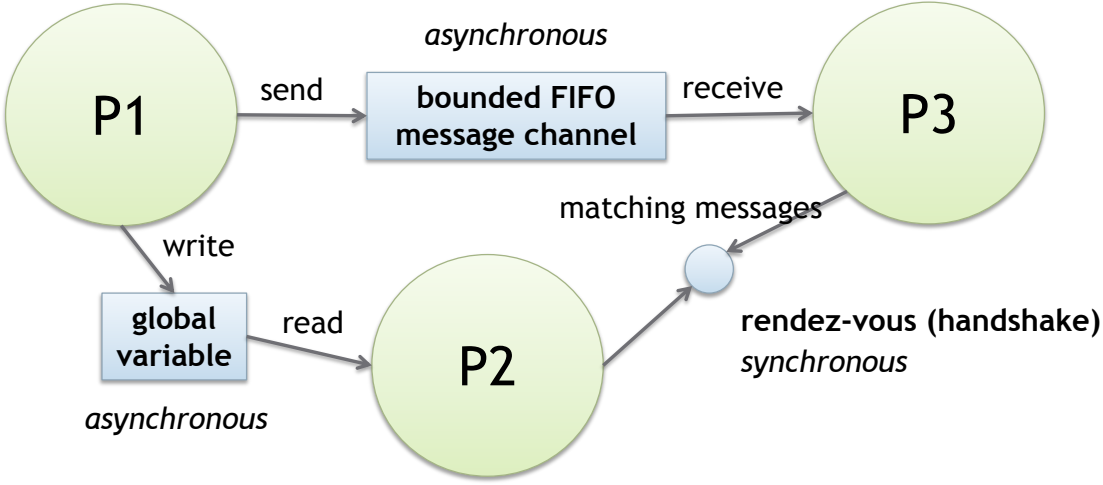
Spin Architecture: Verification



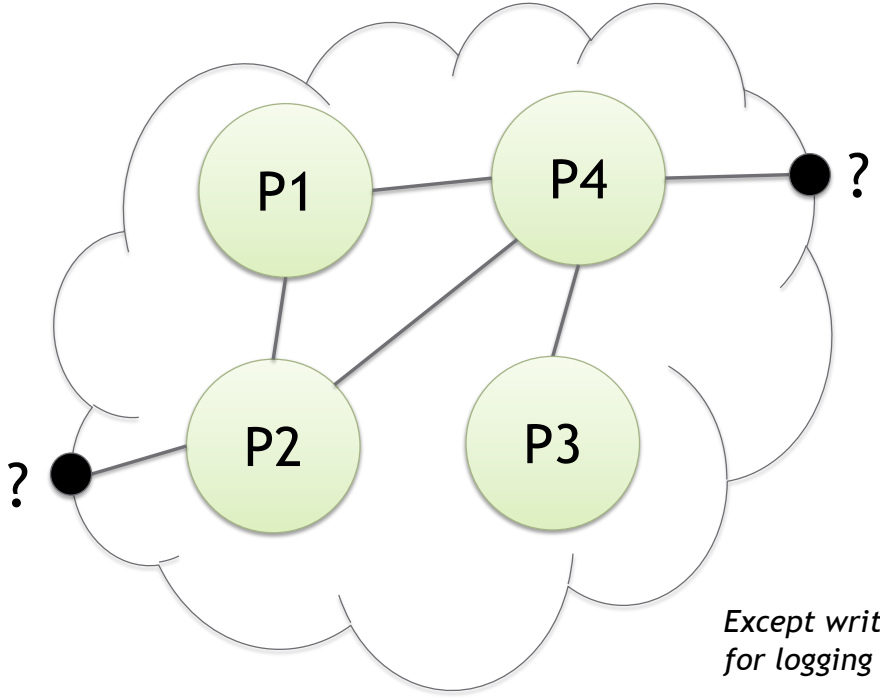
Spin Architecture: Guided Simulation



A Promela model defines a system of communicating processes

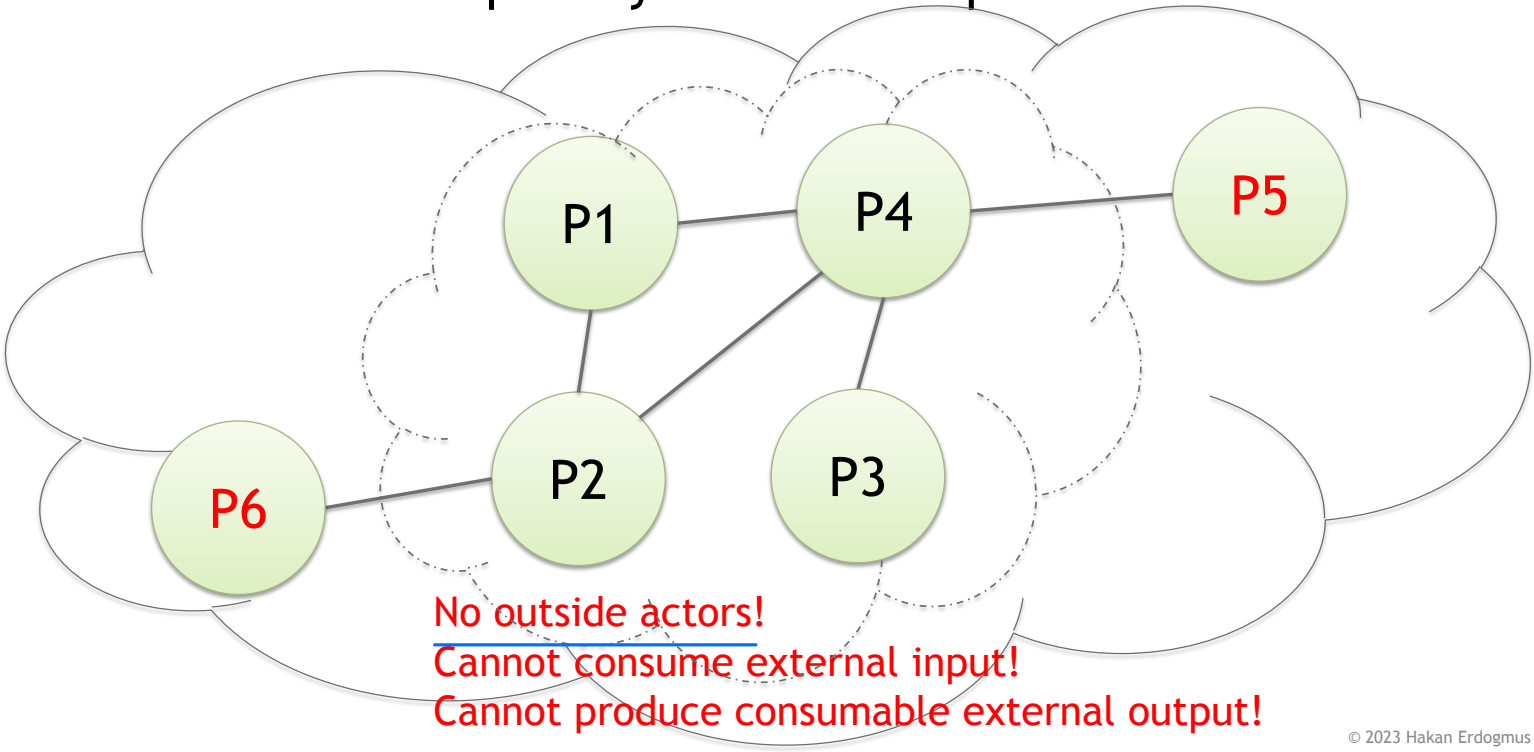


Promela models define a closed system



Except writing to stdout via printf for logging and debugging

Environment, including, users/clients must be explicitly modeled as processes



Each Promela process defines a Finite State Automaton (FSA)

Not same as CFG, but similar/related to it!

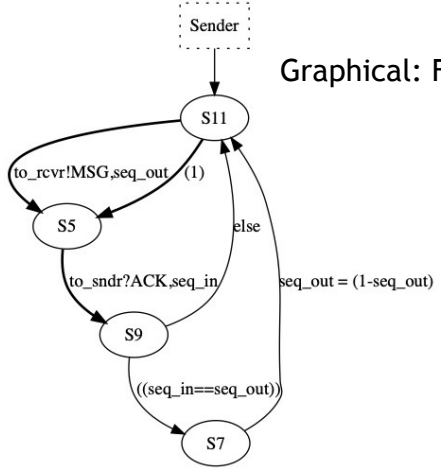
Try it on Mutex example!

- Click on Automata View
- Choose png as format
- Then choose a process p_X or p_Y from the dialog box

Textual: Promela

```
active proctype Sender() {
     bit seq_out = 0;
     bit seq_in = 0;
     do
     ::   if
          :: to_rcvr!MSG(seq_out);
          :: skip;
          fi;
          to_sndr?ACK(seq_in);
          if
          :: seq_in == seq_out ->
             seq_out = 1 - seq_out;
          :: else;
          fi
     od
}
```

Graphical: FSA



Java analogues:

- proctype is like a Class declaration for a class that defines the behavior of a thread
- active (or separate run statement) creates a new instance of a process and starts executing it on a new thread

Promela has *interleaving semantics*

Execution semantics is based on serialization of processes…

- as if we are scheduling them on a single-core
- interleave executable statements from each process
- at any given state, one statement from one active process is randomly selected from the set of all enabled statements from all active processes
- the system then executes that statement and moves to the next state
- exactly as what we have done in the *product automata with Mutex model checking example*…

Promela has few model elements and language constructs

- **type** declarations
- **channel** declarations
- **variable** declarations
- **process** declarations
- **initial** process

Everything must be finite

- no unbounded data
- no unbounded channels
- no processes with infinite number of states
- no unbounded process creation

Otherwise cannot perform verification in finite time!

```
mtype = { MSG, ACK }
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
   // process body
    ...
}

proctype Receiver() {
    // process body
    ...
}

init { // also a process
     // creates other processes
     run Receiver();
     ...
}
```

Source: spin-public/Misc

HelloWorld.pml

makes an instance of this process run automatically

```
// The Promela model for "Hello World"

active proctype Hello() {
    printf("Hello, my pid is: %d\n", _pid);
}

init {
    int lastpid; // a local var
    printf("**** init process, my pid is: %d\n", _pid);
    lastpid = run Hello(); // returns pid of new process instance
    printf("**** last pid was: %d\n", lastpid);
}
```

similar to "this.id"

reserved var: returns process id of current process

```
$ spin -n2 HellowWorld.pml
**** init process, my pid is: 1
**** last pid was: 2
Hello, my pid is: 0
Hello, my pid is: 2
? processes created
```

running spin in simulation mode

Why is this line printed before this line?

How many processes?
Why?

- 3. Hello:0, Init 1, Hello: 2

Try this from command line first.
Then try from SpinRCP -> *Simulation* -> *Random* -> Ok -> *Run*

Process declarations (proctype)

A **process declaration** (**proctype**) consist of

- a name
- a list of parameters
- local variable declarations
- body

Careful! This is a semicolon here if you repeat the type, but when you run Sender ";" becomes comma:

arguments

```
run Sender (ch1, ch2);
```

parameters

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

The body consists of a sequence of statements

Processes can be created in two ways: implicitly and dynamically

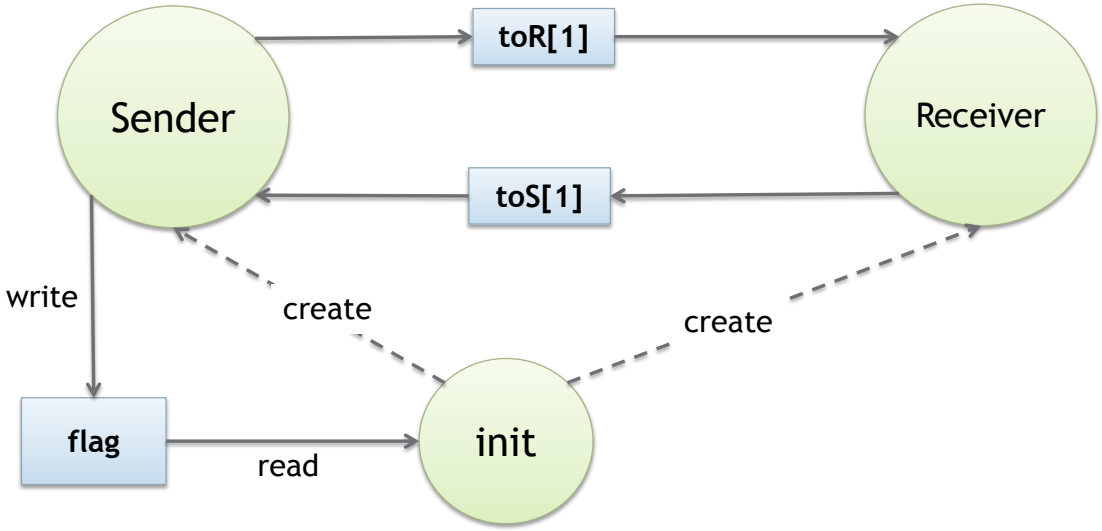
- Process are created
  - using the **run** statement (which returns the **process id**)
  - by the keyword **active** before **proctype**
- Processes can be created dynamically at any point during execution (within any process)
- *Like starting new Java threads in a Java program*

```
proctype Foo(byte x) {
  ... // not running yet
}

init { // runs immediately
   int pid2 = run Foo(2);
   run Foo(27);
}

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

optional number of processes (instances of proctype) to be created

How many process instances here? ○ 6

A Simple Example: SenderReceiver System - Process Diagram



Process diagram: very useful for visualizing the topology of a model after all process instances have started running, *but* shows just topology (structure), no behavior

Source: spin-public/SenderReceiver

SenderReceiver > SenderReceiver.pml

- toS channel can carry one element of MSG or ACK

```
mtype = { MSG, ACK }
chan toS = [1] of { mtype }
chan toR = [1] of { mtype }
// bool flag = false;

proctype Sender() {
   toR!MSG;
   toS?ACK;
   // flag = true;
}

proctype Receiver() {
   toR?MSG;
   toS!ACK;
}

init {
   run Sender();
   run Receiver();
   // assert(flag);
}
```

- Enum
- ? Is equivalent to if (toR == MSG) { xxx }

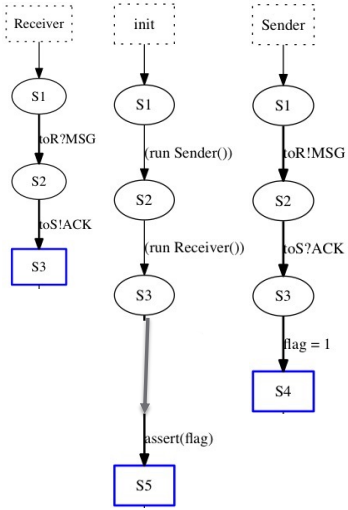
- Create a new file SenderReceiver.pml
- Insert the code on the left
- Syntax Check
- Run a *Simulation* and observe
  - *Random*
  - first try *Run*
  - then try *Step options (at home)*
  - *observe channel (queue) contents*
  - *close simulation* Close
- Uncomment commented statements
- Run a *Verification*
  - *Safety* with *Assertion violations*
- Does it fail? Why?
  - Run a *Guided Simulation*
  - *Check console output for failure trace*

- Will the *Verification* fail each time (unlike in with the test in Mutex in example)?

Source: spin-public/SenderReceiver

SenderReceiver2.pml: Homework

```
mtype = { MSG, ACK }
chan toS = [1] of { mtype }
chan toR = [1] of { mtype }
bool flag = false;

proctype Sender() {
   toR!MSG;
   toS?ACK;
   flag = true;
}

proctype Receiver() {
   toR?MSG;
   toS!ACK;
}

init {
   run Sender();
   run Receiver();
   assert(flag);
}
```



Build this product automaton by hand…

Receiver ⊗ Sender ⊗ init = ??

SenderReceiver3.pml

```
mtype = { MSG, ACK }
chan toS = [1] of { mtype }
chan toR = [1] of { mtype }
bool flag = false;

proctype Sender(chan in, out; mtype msg, ack) {
   out!msg;
   in?ack;
   flag = true;
}

proctype Receiver(chan in, out; mtype msg, ack) {
   in?msg;
   out!ack;
}

init {
   run Sender(toR, toS, MSG, ACK);
   run Receiver(toS, toR, MSG, ACK);
   timeout;
   assert(flag);
}
```

- Parameterize the processes
  - now input and output channels are parameters
- Add timeout to init
  - Timeout wait until all other two processes finished
- Run *Random Simulation* again
- Run a *Verification* again
  - *Safety* with *Assertion violations*
- Same behavior? Why? Why not?
- What's the different in this model?

Variables and scoping

Just like in C (and many other programming languages)

– globals, locals, parameters

```
byte foo, bar, baz;

proctype A(byte foo) {
  byte bar;
  baz = foo + bar;
 }
```

- One **global scope**
- Several **local scopes**
- Local scope overrides global scope when variable names overlap

Variables and types

Basic types

```
bit turn;          [0..1]
bool flag;         [0..1]
byte counter;      [0..255]
short s;           [-2^15-1..2^15-1]
int msg;           [-2^31-1..2^31-1]
```

default initial value of all basic variables is **0**

plus… basic ops on these!

overflow => back to 0

Arrays (index starts at 0)

```
byte a[27];
bit flags[4];
```

Typedef (records) for structured types

```
typedef MyRecord {
    short f1;
    byte f2;
}
MyRecord rr; /* declare a MyRecord var */
rr.f1 = 2334;
rr.f2 = 125;
```

These are not classes or objects!

Enumeration

An enumeration type is used to declare symbolic constants in the global scope

```
mtype = { RED, BLUE, GREEN }
mtype color = RED; // color is an mtype
```

Multiple **mtype** declarations are possible, but they are treated as one big **mtype** declaration, as opposed to different enumeration types: cannot repeat the same constant in different **mtype** declarations

Statements and Executability

- The body of a process consists of statements
  - assignment
  - expression
  - assertion
  - process instantiation
- At any given time during simulation/verification, a statement is either
  - **executable (enabled)**
  - **blocked (disabled)**
- An assignment or an assertion is always executable => these are *actions* or *commands*

Executability of Expressions

- BUT... an **expression** is executable only if it evaluates to a non-zero value

  Therefore: expressions act as guards

```
2 < 3;      always executable
x < 27;     executable only if x < 27 at that point
3 + x;      executable only if x is not equal to −3 at that point
```

This is not an if statement: it is quite different than other programming languages you know!

Executability of Other Statements

- The **skip** statement is always executable: does nothing, only advances the process to a different state (used for readability)
- A **run** statement is only executable if a new process can be created (*remember: the number of processes is bounded*)
- A **printf** statement is always executable (but is ignored during verification)

```
int x; // if initialized, equals 0

proctype Aap()
{
    int y = 1;
    skip;
    run Noot();
    x = 2;
    (x > 2 && y == 1) ->
    skip;
}
```

enabled if some other process (perhaps Noot()?) makes x greater than 2; otherwise blocks until such time

Statement Separators

Statements are separated by `;` or `->`

They have exactly the same meaning and are interchangeable

For readability:

- `;` after declarations, assignments, assertions (non-blocking constructs)
- `->` after expressions/guards (blocking constructs)

```
int x;

proctype Aap()
{
    int y = 1;
    skip;
    run Noot();
    x = 2;
    (x > 2 && y == 1) ->
    skip;
}
```

Deadlocks - Guards > Guards.pml

```
int x;

active proctype Aap()
{
    int y = 1;
    skip;
    x = 2;
    (x > 2 && y == 1) ->
    skip;
}

active proctype Bbp()
{
    // timeout;
    // x = 3;
    skip;
}
```

- Aap will block on this line, never ends
- valid end state
- valid end state

- Run *Verification -> Safety*
  - *Invalid end states*
- Does it fail?
- Why?

- Remove comments (Guards2.pml)
- Run Verification again
- Does it pass?
- Why?

By default, only states before a closing curly brace are valid end states; if the whole system blocks elsewhere, safety verification will interpret that as a deadlock and fail!

Assertions

`assert(<expr>)`

- The **assert** statement is always executable
- If <expr> evaluates to 0, Spin will exit with an error: "assertion violated (<expr>)"
- The assert statement is often used within Promela models to check safety (state) properties (often through a *monitor*, or *spy*, process)

What does this do?

```
active proctype monitor() {
    assert(n <= 3);
}
```

Source: spin-public/Guards

Guards3.pml

```
int x = 0;

active proctype Aap() {
     int y = 1;
     skip;
     x = 2;
     (x > 2 && y == 1) ->
     skip;
}

active proctype Bbp() {
     timeout;
     x = 3;
     skip;
}


active proctype monitor() {
     assert(x > 0);
}
```

- Run *Verification -> Safety*
  - *Assertion violations*
  - *Invalid end states*
- Does it fail? Why?
- Replay the failure trace to see why
  - *Simulation -> Guided*
- How can you make assertion pass? (See: Guards4.pml)

Selection
(Dijkstra's non-deterministic guarded choice)

```
if
   :: choice1 -> stat1.1; stat1.2; stat1.3; … ;
   :: choice2 -> stat2.1; stat2.2; stat2.3; … ;
   :: …
   :: choicen -> statn.1; statn.2; statn.3; … ;
fi;
```

Each of the $\texttt{choice}_i$ *acts as a* ***guard***

Convention/Readability: *use* `->` *instead of* `;` *after a guard*

This is not an ordinary if statement like in other languages: it's nondeterministic and it can block!

Selection
(Dijkstra's non-deterministic guarded choice)

```
if
   :: choice1 -> stat1.1; stat1.2; stat1.3; … ;
   :: choice2 -> stat2.1 ; stat2.2; stat2.3; … ;
   :: …
   :: choicen -> statn.1; statn.2; statn.3; … ;
fi;
```

- If there is at least one $choice_i$ (**guard**) enabled, the **if** statement is executable
- Spin non-deterministically chooses one of the enabled choices during simulation
- During verification, spin tries all enabled choices, as expected
- If no $choice_i$ is executable, the **if** statement blocks (*main difference from Java and C*)

Selection (if-fi) example

```
/*
    assign n a random number
*/

if
:: skip -> n = 0;
:: skip -> n = 1;
:: skip -> n = 2;
:: skip -> n = 3;
fi;
```

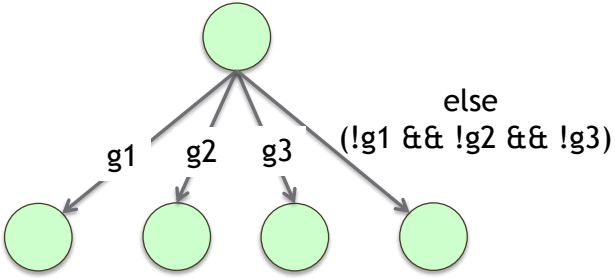
Skips are unnecessary here since assignments are always executable, but can be included for understandability and emphasis to signal an “empty” or “always true” guard

```
/*
  equivalent to this because
  assignment is always
  executable...
*/

if
:: n = 0;
:: n = 1;
:: n = 2;
:: n = 3;
fi;
```

```
/*
  equivalent to this as
  well...
*/

if
:: true -> n = 0;
:: true -> n = 1;
:: true -> n = 2;
:: true -> n = 3;
fi;
```

Selection with else guard

The **else** guard becomes executable if none of the other guards are enabled

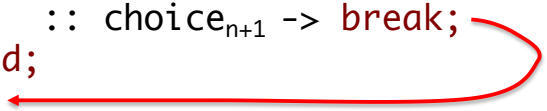


non-deterministic branching with else

```
if
:: (n % 2 != 0) -> n = 1;
:: (n >= 0) -> n = n - 2;
:: (n % 3 == 0) -> n = 3;
:: else -> skip; // do nothing
    // ok to omit skip above
fi;
```

Repetition (do-od) is like selection (if-fi), but loops indefinitely until broken

```
do
   :: choice1 -> stat1.1; stat1.2; stat1.3; … ;
   :: choice2 -> stat2.1; stat2.2; stat2.3; … ;
   :: …
   :: choicen -> statn.1; statn.2; statn.3; … ;
   :: choicen+1 -> break;
od;
```



- The (always executable) **break** statement exits a **do** loop and transfers control to the end of the loop
- **break** is optional
  - **break** can be anywhere in the loop
  - can have multiple **breaks**
  - **break** can be absent altogether, causing an infinite loop (sometimes exactly what we want)

Repetition (for statement)

Included for convenience!

See Promela manual:

http://spinroot.com/spin/Man/promela.html

Source: spin-public/TrafficLight

Repetition example: TrafficLight > TrafficLight2.pml

```
mtype = { RED, GREEN, ORANGE }

active proctype TrafficLight() {
    mtype status = GREEN;

    do
    :: (status == GREEN) -> status = ORANGE;
    :: (status == ORANGE) -> status = RED;
    :: (status == RED) -> status = GREEN;
    od;
}
```

- Run *Random Simulation*
- What's happening?
- Must explicitly stop it, and close the simulation!

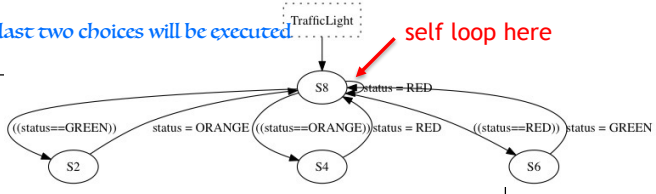
Nondeterminism: TrafficLight3.pml

```
mtype = { RED, GREEN, ORANGE }

active proctype TrafficLight() {
    mtype status = GREEN;

    do
    :: (status == GREEN) -> status = ORANGE;
    :: (status == ORANGE) -> status = RED;
    :: (status == RED) -> status = GREEN;
    :: status = RED; // extra choice
    od;
}
```

Generate FSA
- *Automata View*
- *Turn off statement merging*

added extra choice

If status == RED, either one of the last two choices will be executed nondeterministically.



- Run *Random Simulation*
- What's happening?
- Run *Verification -> Safety > Assertion Violations & Invalid end states*
- Does *Verification* end? Why?

Conditionals and Loops - more examples

```
if
:: in?ACK -> skip;
:: in?MSG -> received++;
   out!ACK;
:: else -> out!ERR;
fi;
```

```
do
:: (received < MAX) -> in?MSG;
   received++; out!ACK;
:: received == MAX -> break;
od;
```

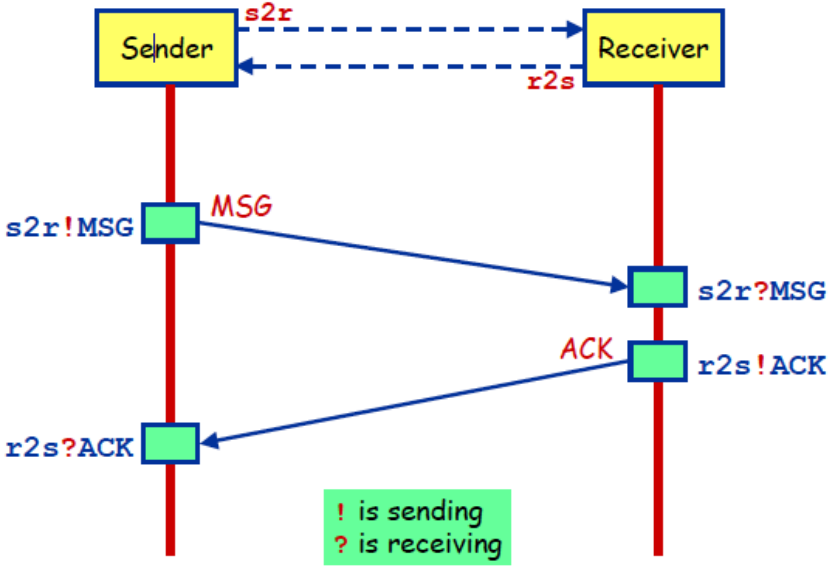
When reach the maximum, break

Jump

The infamous **goto**…

The above is equivalent to do-od (without a break)

Inter-process communication can be via messages sent through channels



You can visualize channel communication in SpinRCP in simulation mode:
use MSC Viewer tab

Message Sequence Chart (MSC) – resembles a UML sequence diagram

Ruys, 2002

Promela channels can be async or sync

- message passing (async, dim > 0) Can hold more than one slots in the channel, so async.
- rendez-vous synchronization (handshake, sync, dim = 0)

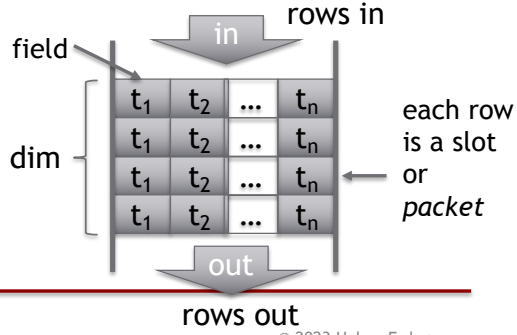
```
chan <name> = [<dim>] of { <t1>,<t2>, … <tn> }
```

max size (capacity) of channel:
# of slots

data types of fields in each slot
(order is important)

```
chan c = [1] of { bit }
chan sync = [0] of { mtype }
chan toR = [2] of { mtype, bit }

/* an array of channels of size 1 each */
chan line[2] = [1] of { mtype, MyRecord }
```

This one is sync



A Promela channel behaves as a FIFO-buffer (for dim > 0, async)

! *sending*: putting a message into a channel

ch ! <expr$_1$>, <expr$_2$>, … <expr$_n$>;

The statement might get blocked if ch capacity is smaller than n in this case. But when receiver retrieves from ch, then it will no longer be blocked

- The inferred types of <expr$_i$> should match the types in the channel declaration
- A send statement is executable if the channel is not full

Promela channels: receiving

? *receiving*: retrieving a message from a channel

ch ? <var$_1$>, <var$_2$>, … <var$_n$>;
ch ? <const$_1$>, <const$_2$>, … <const$_n$>;

- <var>: *to receive values* - message field is fetched from the channel and its value is is stored in the variable
- <const>: *to match values* - message field is fetched from channel only if its value matches the constant
- <var> and <const> can be mixed in the same receive statement

receiving is an ALL or NONE op!

Promela channels: enabled rendez-vous (sync):

Two processes move at the same time when synchronization is possible!

```
chan myCh = [0] of { bit, byte }
```

Wow

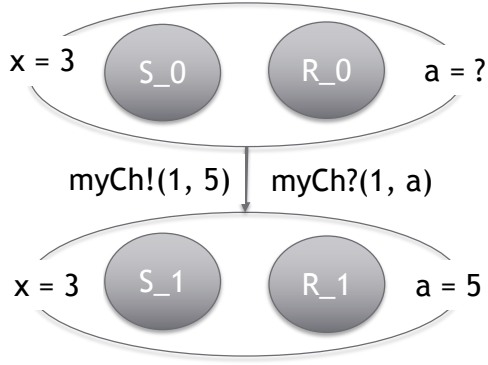
Process S

```
myCh !   1,   x + 2
```

match ↕ pass value

Process R

```
myCh ?   1,   a;
```



- const 1 in *R* must match constant 1 in *S for synchronization to occur*
- var a of *R* is assigned the value of x + 2 from *S when synchronization occurs*
- After synchronization, both processes simultaneously advance to their respective next states (no "sending before receiving").

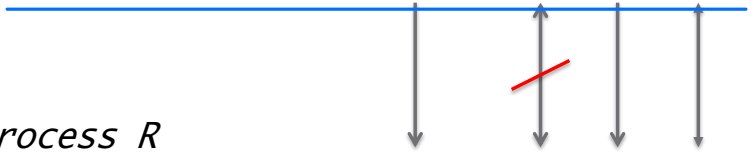
S and R moves to the next states at the same instant

Promela channels: disabled rendez-vous

```
chan myCh = [0] of { int, byte, int, mytpe }
```

Process S

```
myCh ! x + 1, 100, y, ACK;
```



Process R

```
myCh ?   a,   200, b, ACK;
```

- variable *a* in *R* can be assigned the value of x + 1 from *S*
- const 100 in *R* cannot be matched to constant 200 in *S*
- var b in *R* can be assigned the value of y from *S*
- mtype ACK in *R* matches mtype ACK in *S*

→ rendez-vous between *S* and *R* is not executable (blocks): no values are passed!

All or Nothing !

Convention for message passing syntax

First message field often specifies message type (constant/mytpe)

```
qname!const(expr2, expr3)  // always preferred
```

Same as: `qname ! const, expr2, expr3`

```
qname?const(var2, var3)    // always preferred
```

Same as: `qname ? const, var2, var3`

This is like sending a packet of type const that stores two data values expr2 and expr3

Full/empty channel blocks send/receive

- Message channels are first-in, first-out (FIFO) *by default*

- Send is executable only when the channel is not full

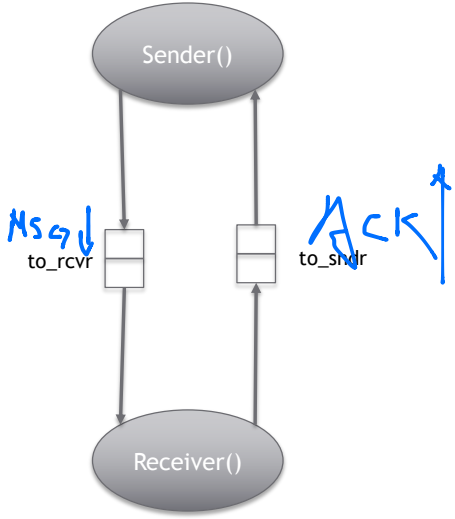
- Receive is executable only when the channel is not empty

Otherwise, it will be blocked and wait

Source: spin-public/ABP

Simple Alternating Bit Protocol: ABP > AltBit.pml (no loss)

```
mtype = { MSG0, MSG1, ACK0, ACK1 }

chan    to_sndr = [2] of { mtype }
chan    to_rcvr = [2] of { mtype }

active proctype Sender() {
    again:
    to_rcvr!MSG1;
    to_sndr?ACK1;
    to_rcvr!MSG0;
    to_sndr?ACK0;
    goto again;
}

active proctype Receiver() {
    again:
    to_rcvr?MSG1;
    to_sndr!ACK1;
    to_rcvr?MSG0;
    to_sndr!ACK0;
    goto again;
}
```



Source: spin-public/ABP

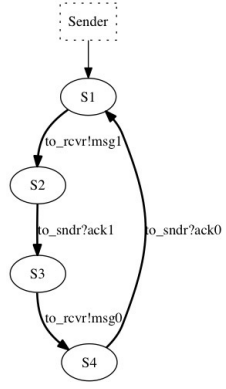
Simple Alternating Bit Protocol: ABP > AltBit.pml (no loss)

- MSG and ACK have alternating seq. no.s
- If ACK seq. no. doesn't match MSG seq. no., something is wrong (e.g., MSG was lost)!

```
mtype = { MSG0, MSG1, ACK0, ACK1 }

chan    to_sndr = [2] of { mtype }
chan    to_rcvr = [2] of { mtype }

active proctype Sender() {
    again:
    to_rcvr!MSG1;
    to_sndr?ACK1;
    to_rcvr!MSG0;
    to_sndr?ACK0;
    goto again;
}

active proctype Receiver() {
    again:
    to_rcvr?MSG1;
    to_sndr!ACK1;
    to_rcvr?MSG0;
    to_sndr!ACK0;
    goto again;
}
```

- Run *Random Simulation*
- Observe
- Run *Safety Verification*
  - *Invalid end states*
- Does verification pass?
- What does it mean?
- Can we parameterize this?
- Can we model retransmission when ACK seq. no. doesn't match?
- Can we model message losses?

Generate FSA *Automata View*



Source: spin-public/ABP

Parameterized Alternating Bit Protocol: ABP > AltBit1.pml (no loss)

Parameterize sequence number of MSG and ACK

Retransmit MSG if seq. no.s mismatch

```
mtype = { MSG, ACK }
chan  to_sndr = [2] of { mtype, bit }
chan  to_rcvr = [2] of { mtype, bit }

active proctype Sender() {
      bit seq_out = 0;
      bit seq_in = 0;
      // obtain first message
      do
      :: to_rcvr!MSG(seq_out) ->
         to_sndr?ACK(seq_in);
         if
         :: (seq_in == seq_out) ->
            // accept new message
            seq_out = 1 - seq_out;  Flip the seq_out
         :: else
            // retransmit same message
         fi;
      od;
}

active proctype Receiver() {
      bit seq_in;
      do
      :: to_rcvr?MSG(seq_in) ->
         to_sndr!ACK(seq_in);
      od;
}
```

- Run *Random Simulation*
- Observe
- Run *Safety Verification*
  - *Invalid end states*

- Retransmissions, but…
- Still lossless
- Can we model random message losses?
  - sndr loses a message
  - rcvr loses a message

note that we are not modeling the payload: it's irrelevant (if message with type and seq-no arrives, so does the payload)

Parameterized Alternating Bit Protocol: AltBit2.pml (lossy, with timeout to recover from message loss)

```
mtype = { MSG, ACK };

chan   to_sndr = [2] of { mtype, bit };
chan   to_rcvr = [2] of { mtype, bit };

active proctype Sender() {
       bit seq_out = 0;
       bit seq_in = 0;
       // obtain first message
       do
       ::  if
           :: to_rcvr!MSG(seq_out); // send message
           :: skip; // or loose it
           fi;
           to_sndr?ACK(seq_in);
           if
           :: (seq_in == seq_out) ->
              // accept new message
              seq_out = 1 - seq_out;
           :: else; // retransmit same message
           fi;
       od;
}

active proctype Receiver() {
       bit seq_in = 1; // important for forcing wrong ACK with loss
       do
       :: if
          :: to_rcvr?MSG(seq_in);  // receive msg
          :: timeout; // recover from msg loss
          fi;
          if
          :: to_sndr!ACK(seq_in); // send ack
          :: skip; // or loose it
          fi;
       od;
}
```

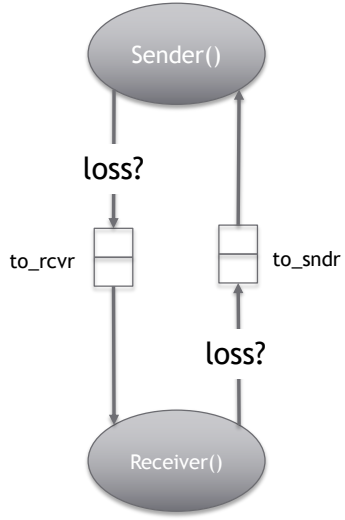
- Run *Random Simulation*
- Observe
- Run *Safety Verification*
  - *Invalid end states*

- Comment out “:: timeout”
- Repeat
- What happened? Why?

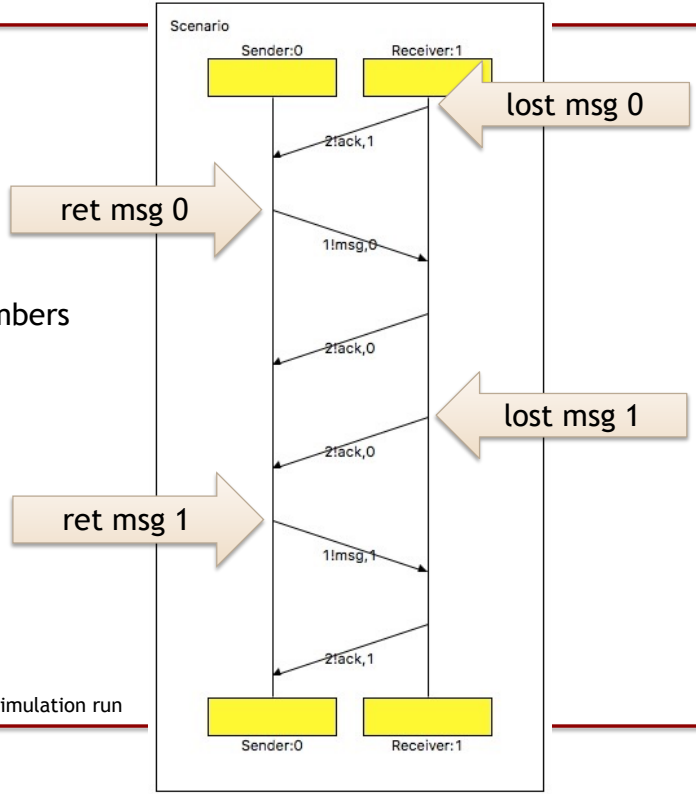
timeout: behaves like **else** guard, but enabled only if no process can make progress

- **else:** local escape
- **timeout:** global escape

If the receiver realizes that something went wrong(sender didn't receive my ack), then I call timeout to stop the whole system (all processes) for a while.



Example: ABP Message Sequence Chart



Mismatch of sequence numbers triggers retransmission by sender

Generated by SpinRCP simulation run

Controlling atomicity

d_step

atomic

A sequence of statements can be made fully atomic

`d_step { stat1; stat2; ... statn; }`

- useful for performing intermediate computations in a single indivisible transition when this doesn't affect the behavior of the whole system
- useful for bookkeeping of observation variables without increasing number of states

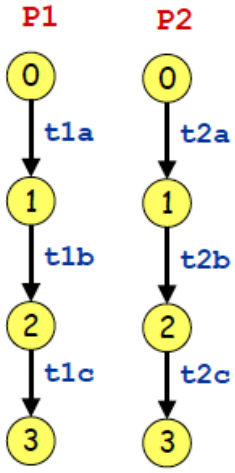
- no intermediate states are generated and stored
- may *only* contain deterministic steps
- run-time error if $stat_i$ (i > 1) blocks
- should not have guards or non-deterministic choices

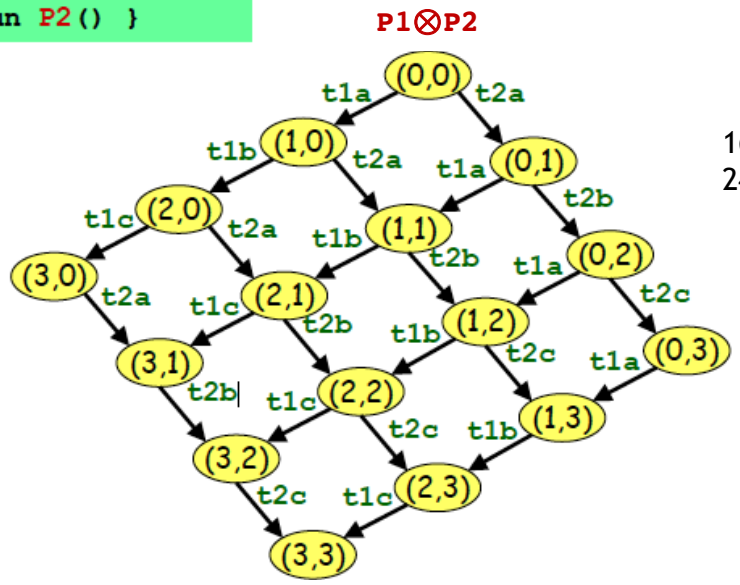
```
:: Rout?i(v) ->
   d_step {
      k++;
      e[k].ind = i;
      e[k].val = v;
       i = 0; v = 0;
   }
```

A weaker version is the atomic construct: see Appendix

With no atomicity, we generate lots of states during verification

```
proctype P1() { t1a; t1b; t1c }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```





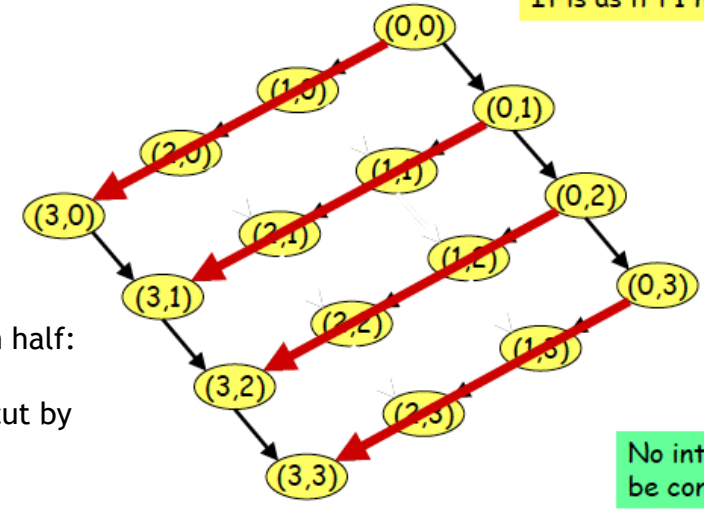
16 states
24 transitions

d_step may reduce number of states and transitions significantly, …

```
proctype P1() { d_step {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

It is as if P1 has only one transition…

P1⊗P2



… but it may eliminate intended concurrency (be careful!)

States are cut in half:
8 states
Transitions are cut by less than half:
10 transitions

No intermediate states will be constructed.

C preprocessor (cpp) directives can be used in Promela models

All cpp directives (#define, #ifdef, #include, …) can be used in Promela:

Constants

```
#define MAX 4
```

Macros

```
#define RESET_ARRAY(a) \
    d_step { a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; }
```

Conditional model fragments

```
#define LOSSY 1
…
#ifdef LOSSY
active proctype Daemon() {
   // steal messages
}
#endif
```

Caution: these are macros (syntactic substitution), not procedures –cpp expands these before compilation!

Inlined macros

- Promela also has its own macro-expansion feature: the `inline` construct
  - produces more meaningful error messages compared to `#define`
  - cannot be used in place of an expression; only used as a statement
  - all variables should be declared somewhere else

```
inline  initArray(a) {
    d_step  {
        i = 0;
        do
        ::  i < N -> a[i] = 0;
           i++;
        ::  else -> break
        od;
        i = 0;
    }
}
```

Note that this is purely syntactic, not a procedure!

More efficient and better error messaging than cpp directives

No built-in procedures/functions/methods in Promela!

... but these are possible to emulate using processes and message passing!

Review Appendix: includes cautionary uses and other features that may be needed in assignments

Key Messages

Spin is a simple but powerful model checking tool for modeling and verifying finite-state systems defined in Promela

- systems defined as communicating processes
- on-the-fly verification: explore state state space and check properties at the same time
- C-like syntax
- many optimizations
- supports both synchronous and asynchronous inter-process communication
- concurrency is modeled by interleaving of enabled transitions: only one process advances at a time

Appendix

You may need the material in this Appendix for your labs and assignments

More on Promela processes

A **process**

- is an instance of **proctype** definition
- there may be several processes of the same **proctype**
- executes concurrently (via interleaving semantics) with all other processes
- communicate with other processes using
  - **global** (shared) **variables**
  - synchronous or asynchronous **channels**
- has its own local state composed of
  - a process counter (location within the **proctype**)
  - values of **local variables**
- makes no assumptions with regard to its or other processes' speeds of execution

Process execution
(more on interleaving semantics)

Promela processes execute *concurrently*

- scheduling is non-deterministic
- all statements are atomic
- multiple statements can be enabled at any time
- statement choice (which to execute next) is non-deterministic
- processes are interleaved
  - *exception*: rendez-vous communication

Source: spin-public/Mutex

Dekker.pml: A mutex algorithm (Dekker's algorithm [1962])

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1; // I want access
      turn = B_TURN; // B's turn after I'm done
      (y == 0 || (turn == A_TURN)) ->
      mutex++; // in critical section
      mutex--; // out of critical section
      x = 0; // I'm done
}

active proctype B() {
      y = 1; // I want access
      turn = A_TURN; // A's turn after I'm done
      (x == 0 || (turn == B_TURN)) ->
      mutex++; // in critical section
      mutex--; // out of critical section
      y = 0; // I'm done
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Run *Safety Verification*
  - *Assertion violations*
- Does it pass?

This is a fix for the progress problem we had earlier with the initial Mutex example!

[Dekker, 1962] First software-only solution to the mutex problem (for two processes)

Source: spin-public/Mutex

Dekker2.pml

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
// mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1;
      // turn = B_TURN;
      y == 0 // || (turn == A_TURN) ->
      mutex++; // in critical section
      mutex--;
      x = 0;
}

active proctype B() {
      y = 1;
      // turn = A_TURN;
      x == 0 // || (turn == B_TURN) ->
      mutex++; // in critical section
      mutex--;
      y = 0;
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Remove turn variable
- Run *Safety Verification*
  - *Assertion violations*
- Does it pass? Why?

Source: spin-public/Mutex

Dekker2.pml

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
// mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1;
      // turn = B_TURN;
      y == 0 // || (turn == A_TURN) ->
      mutex++; // in critical section
      mutex--;
      x = 0;
}

active proctype B() {
      y = 1;
      // turn = A_TURN;
      x == 0 // || (turn == B_TURN) ->
      mutex++; // in critical section
      mutex--;
      y = 0;
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Remove turn variable
- Run *Safety Verification*
  - *Assertion violations*
- Does it pass? NO!
- This becomes the same mutex algorithm we tried earlier in course (but instead of a busy-wait, we have a simple guard, so we should get a deadlock instead of a livelock)
- Run *Guided Simulation* with failure trace to check your hunch

Source: spin-public/Misc

SquareCube.pml: Procedures can be modeled as processes exchanging messages in a request-response protocol

```
mtype = { SQUARE, CUBE, RETURN }
chan toS = [0] of { mtype, int }
chan toR = [0] of { mtype, int }

active proctype Client() {
   int y;
   toR!SQUARE(2);
   toS?RETURN(y);
   assert(y == 4);
}

active proctype Server() {
   int x;
   if
   :: toR?SQUARE(x) ->
      toS!RETURN(x*x);
   :: toR?CUBE(x) ->
      toS!RETURN(x*x*x);
   fi;
}
```

- Run *Verification -> Safety*
  - *Assertion violations*
- Does it pass? Why?

`len(qname)` returns the number of messages/slots currently stored in channel *qname*

- `len` is always executable if used inside an expression

```
len(ch) + 1
```

- `len` used as a guard is executable only if the channel is non-empty (blocks if `len` evaluates to 0)

slot

```
:: len(ch) -> …
```

The receiving end of a channel can be polled

Can poll a receive to test if it is executable

```
qname?[ack,var]
```

- *Evaluates to* 1 *if reception is enabled*; 0 *if not*
- *The receive is not executed*

Be careful about timeout!

- Promela does not have real-time features
- **timeout** models a global timeout
- **timeout** becomes executable if there is no other process in the system which is executable

Caution!

- **timeout** may mask deadlocks
- **timeout** may interfere with a monitor process that checks a global assertion (invariant)
- Used only to model timeout behavior intentionally, not to make a verification pass to pass an assignment

Partial atomicity: can force execution of successive statements as an atomic sequence

`atomic { stat1; stat2; ... statn; }`

- is executable if $stat_1$ is executable
- blocks *temporarily* if a successive statement is blocked
- statements can be non-deterministic

More flexible than `d_step`

```
proctype P(bit i) {
    atomic {
        flag != 1 -> flag = 1;
    }
    mutex++;
    // in critical section
    mutex--;
    flag = 0;
}
```

Beware of race conditions with guards

Consider the following


```
qname?[msgtype]      -> qname?msgtype;

(len(qname) < MAX) -> qname!msgtype;
```

Due to possible race conditions or interleavings by other processes: statement is not necessarily executable after *guard passes*

Use of atomic and d_step

atomic and **d_step** can be used to lower the number of states of the model by preventing unnecessary interleavings

But use with caution: they may change the intended behavior drastically (via underabstraction) and cause a property to pass or fail when it should not!

Don't use them just to pass a verification in an assignment!

Modeling interrupts and exceptions

```
{ <stats> } unless { guard; <stats> }
```

escape sequence

Statements in <stats> are executed until the first statement (guard) in the escape sequence becomes executable

resembles exception handling in languages like Java

```
proctype MicroProcessor() {
    {
      ...
      // execute normal instructions
    }
    unless {
        port?INTERRUPT ->
        ...
    }
}
```

Promela statements: executability rules

• `skip`	always executable
• `assert(<expr>)`	always executable
• `expression`	executable if not zero
• *`assignment`*	always executable
• `if-fi`	executable if at least one guard is executable
• `do-od`	executable if at least one guard is executable
• `break`	always executable (exits do-statement)
• *`send`* (ch`!`)	executable if ch is not full
• *`receive`* (ch`?`)	executable if ch is not empty and constant fields match
• *`rendez-vous send-receive`*	executable together if fields are matchable

How powerful is Spin?

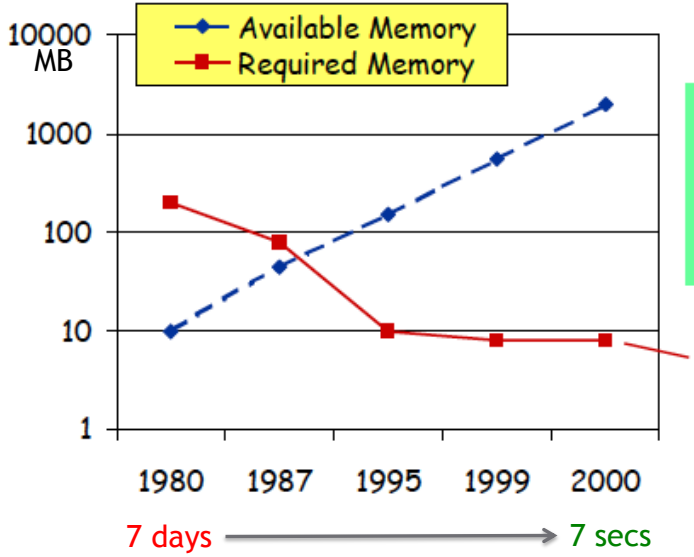
Curiosity spacecraft flight software (the Mars Science Laboratory mission) [Holzmann 2014]

- 45,000 lines of C code abstracted as 1600 lines of Promela model
- Subtle rare concurrency flaws found and fixed

Reported in reading: “The Mars Code,” Communications of the ACM, Feb 2014

How powerful is Spin and how far has model checking come along?

Similar progress in other model checking tools



1980: pan
1987: bitstate hashing *
1995: partial order reduction *
1999: minimised automaton

memory requirements to (fully) verify Tpc

Tpc: "The Phone Company," a small Bell Labs phone system

Ruys, 2002

Credits

- T. C. Ruys, Spin Beginner Tutorial, 2002 Spin Workshop
- G. Holzmann, The Spin Model Checker, Addison Wesley, 2004
- S. Chaki, Introduction to Spin and Promela, Parts 1-3, CMU Lecture Notes, 2010-2012

Spin resources

- Spin web site: http://spinroot.com/
  - Everything you want to know about Spin
    - Downloads
    - Documentation
    - Tutorials
    - Links to 3rd-party tools
- Promela quick reference: http://spinroot.com/spin/Man/Quick.html
- SpinRCP: http://lms.uni-mb.si/spinrcp/
  - Standalone Eclipse Rich-Client-Platform GUI
- Readings
  - See under *Supplements & Readings* on Canvas
- Book
  - Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley

Main Promela Features

Haves

- Communicating processes
- C-like syntax: logical and numerical operations mimic those of C
- Control structures to express
  - Selection (non-deterministic choice) (`if-fi`)
  - Repetition (`do-od`)
  - Jumps (`goto, break`)
- Communication via
  - bounded message channels (*asynchronous* or *rendez-vous*)
  - shared variables
- Embedded LTL properties (more on this later)
- Assertions

Have-nots

- Floating point data type
- Pointers
- Input
- Ability to express real-time behavior