PROMELA

Formal Methods in SE, 07.03.2020

PROcess MEta LAnguage

- PROMELA is **not** a programming language.
- PROMELA is based on Dijkstra's *guarded command language*.
- PROMELA provides:
  - non-deterministic control structures,
  - primitives for dynamic creation of concurrent processes and
  - primitives for interprocess communication.

PROcess MEta LAnguage

- PROMELA is **not** a programming language.
- PROMELA is based on Dijkstra's *guarded command language*.
- PROMELA provides:
  - non-deterministic control structu
  - primitives for dynamic creation o processes and
  - primitives for interprocess comm

NO:

- **functions with return values,**
- **expressions with side-effects,**
- **data and function pointers.**

Types of Objects

We will work with three basic types of objects:

- processes,
- data objects and
- message channels.

Types of Objects

We will work with three basic types of objects:

- processes,
- data objects and
- message channels.

and also **labels**

Processes

Process Initialization

```
active [2] proctype main() {
        printf("my pid is: %d\n", _pid)
}
```

- **active** instantiates 2 processes of the type that follows.
- **proctype** denotes that *main* is a process type.
- *main* is an identifier for the process code block.
- **NB!** ‘;’ is a statement separator, rather than terminator.

Process Initialization

```
active [2] proctype main() {
        printf("my pid is: %d\n", _pid)
}
```

- **active** instantiates 2 processes of the type that follows.
- **proctype** denotes that *main* is a pr
- *main* is an identifier for the process
- **NB!** ';' is a statement separator, rat

```
$ spin test1.pml
            my pid is: 1
         my pid is: 0
2 processes created
```

Process Initialization

```
init {
        printf("Hello, world!\n")
}
```

- **init** is a process that is active in the initial system state.
- **init** is commonly used to initialize a system.
- **init + active** processes ➡️ instantiated in the declaration order.

Process Initialization

```
init {
      printf("Hello, world!\n")
}
```

- **init** is a process that is active in the initial system state.
- **init** is commonly used to initialize a
- **init + active** processes ➡️ instantia
  order.

```
$ spin test1.pml
      Hello, world!
1 process created
```

Process Initialization

```
proctype you_run(byte x) {
      printf("x = %d, pid = %d\n", x, _pid);
      run you_run(x + 1) // recursive call
}
init {
      run you_run(0)
}
```

- **run** creates the process when the instruction is processed.
- **run** allows for input parameters.

Process Initialization

```
proctype you_run(byte x) {
      printf("x = %d, pid = %d\n", x, _pid);
      run you_run(x + 1) // recursive call
}
init {
      run you_run(0)
}
```

- **run** creates the process when the processed.
- **run** allows for input parameters.

```
$ spin test1.pml
           x = 0, pid = 1
               x = 1, pid = 2
                   x = 2, pid = 3
. . .
spin: too many processes (255 max)
      timeout
. . .
255 processes created
```

Process Initialization

```
active proctype you_run(byte x) {
      printf("x = %d\n", x)
}
```

- no parameter can be given to **init** or **active** processes.
- if so, parameters of active processes are 0 by default.

Process Initialization

```
active proctype you_run(byte x) {
      printf("x = %d\n", x)
}
```

- no parameter can be given to **init** or **active** processes.
- if so, parameters of active process

```
$ spin test1.pml
      x = 0
1 process created
```

Data Objects

Basic Types

Type	Range
bit	0, 1
bool	false, true
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$

- A byte can be printed as a character with the %c.
- There are no floats or strings.

Variable Declarations

`bit x,y;`	two single bits, initially 0
`bool turn = true;`	boolean value, initially true
`byte a[12];`	an array of size 12, all elements are initially 0
`byte a[3] = {'h', 'i', '\0'};`	byte array emulating a string of 3 symbols
`chan m;`	uninitialized message channel
`mtype n;`	uninitialized mtype variable
`short b[4] = 89;`	an array of size 4, all elements are initially 89
`int int = 67;`	integer, initially 67
`unsigned v : 5;`	unsigned stored in 5 bits
`unsigned v : 3 = 5;`	value range 0..7, initially 5

- All variables are initialized to 0 by default.
- Array indices start at 0.

Expressions

Assignment: `x = 5;`

Operators:

Arithmetic:	+	-	*	/	%	++	--
Relational:	>	>=	<	<=	==	!=	
Logical:	!	&&	||				
Bitwise:	&	|	^	>>	<<		

Variable Scope

Spin (version 6+) has three levels of scope:

- **global**: declaration is outside of all process bodies,
- **local**: declaration within a process body,
- **block**: declaration within a nested block.

Variable Scope

```
init {
        int x;
        {
            int y;
            printf("x = %d, y = %d\n", x, y);
            x++; y++
        }
        printf("x = %d, y = %d\n", x, y)
}
```

Variable Scope

```
init {
        int x;
        {
            int y;
            printf("x = %d, y = %d\n", x, y);
            x++; y++
        }
        printf("x = %d, y = %d\n", x, y)
}
```

local variable of `init`

outside of scope!

Enumerations

```
mtype = {red, green, yellow};
mtype light = red;
```

- **`mtype`** defines symbolic values (similar to an enum declaration in C).
- indexed from 0.

Executability & Guarded statements

Process Execution

- Processes execute **concurrently**.
- Processes are scheduled **non-deterministically**.
- Processes are **interleaved**.

- Each process has its own local state: **_pid** & **variables**.
- Processes can communicate via: **channels** or **shared variables**.

Statements

- Every basic statement is **atomic**.
- Every statement is either **executable** or **blocked**.
- Basic statements are composed into *compound statements*.

Statements

Always executable:

- print statements `printf();`
- assignments `x = 5;`
- **`skip;`**
- assertions `assert(x >= 2);`
- **`break;`**

Statements

Not always executable:

- **`run`** statement is executable only if there are less than 255 processes alive;
- **`timeout`** statement is executable only if there is no other executable process expressions.

Statements

- An expression is executable iff it evaluates to true (i.e. non-zero):
  - **`(5 < 30)`** always executable;
  - **`(x < 30)`** blocks if x is not less than 30;
  - **`(x + 30)`** blocks if x is equal to -30.
- Expressions must be side-effect free (**`b = c++`** is not valid).

Guarded Statements

selection:

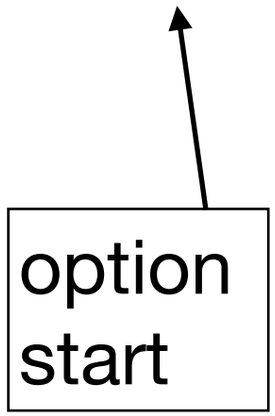
```
if
   :: c_0  -> s_0; . . .
   . . .
   :: c_n  -> skip;
   :: else -> s_e; . . .
fi
```

repetition:

```
do
   :: c_0  -> s_0; . . .
   . . .
   :: c_n  -> s_n; break;
   :: else -> s_e; . . .
od
```

- **{s_i; . . . }** executed only if **c_i** is executable.
- if more than one **c_i** is executable, then an executed branch is chosen non-deterministically.
- if no **c_i** is executable, the **else** branch is executed (if present).
- **break** may be used to exit from a loop.
- **skip** has no effect.

```
:: c_0 -> s_0; . . .; s_n;
```



guard

```
:: c_0 -> s_0; . . .; s_n;
```

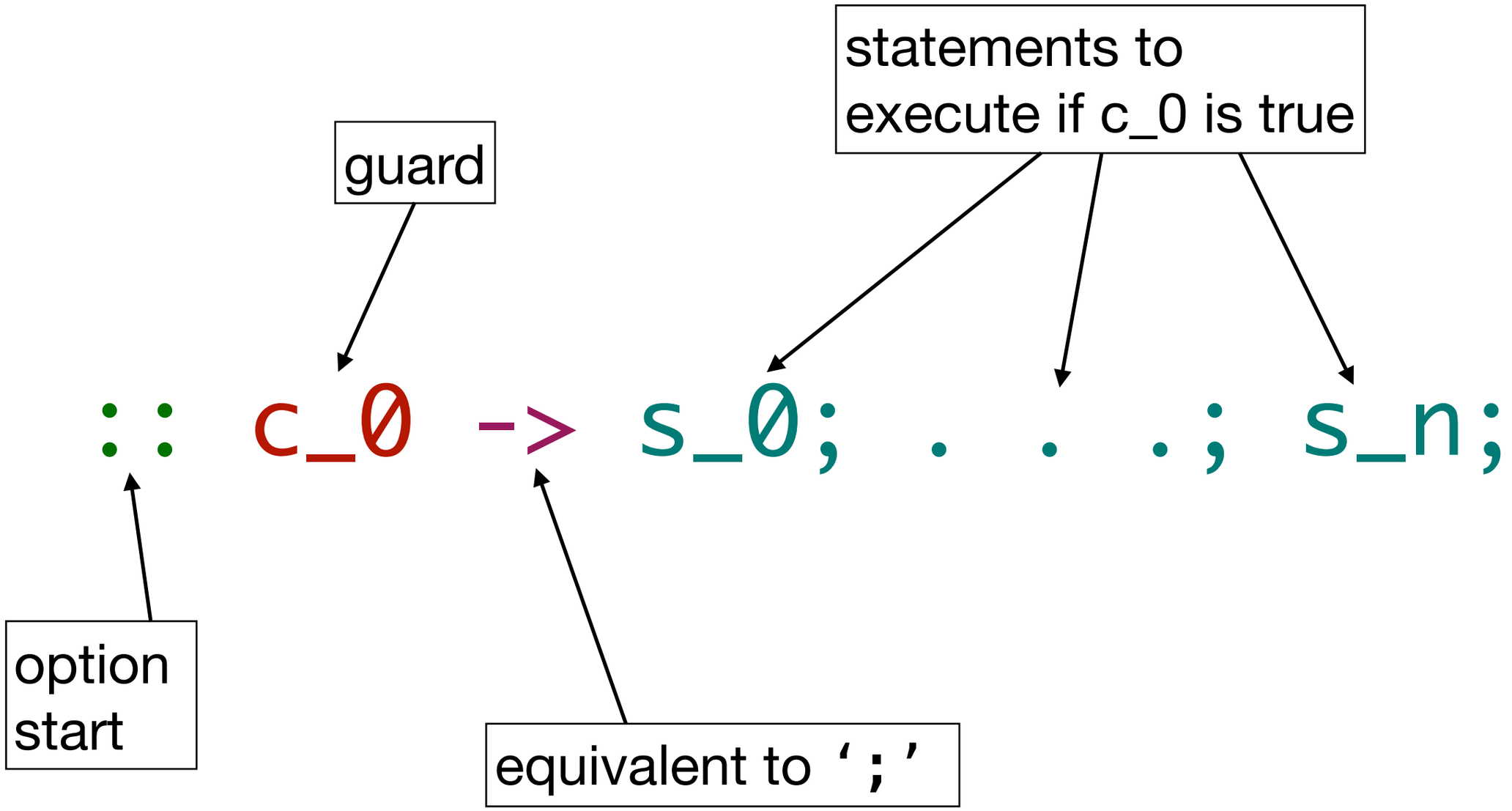
option start

guard

```
:: c_0 -> s_0; . . .; s_n;
```

option start

equivalent to ;


guard
statements to
execute if c_0 is true
:: c_0 -> s_0; . . .; s_n;
option
start
equivalent to ‘;’

Guard Examples

Simple selection

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

Guard Examples

Simple selection with **`else`**

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

NB! else is executable if no other statement *in the same process* is executable

Guard Examples

Guards may overlap ➡️ non-deterministic selection

```
a = 4; b = 4; x = 7;
if
:: (a == b) -> b = 3
::  b = 3
:: (x >= 5) -> b = 3

fi
```

NB! Any statement can be guard

Guard Examples

Greatest Common Divisor

```
do
   :: (m > n)     ->    m = m - n
   :: (m < n)     ->    n = n - m
   :: (m == n)    ->    break
od;

printf("GCD = %d\n", m)
```

Guard Examples

`skip` statement

```
do
    :: !x -> break
    :: else -> skip
od
```

Guard Examples

`skip` statement as a guard

```
if
   :: (x == 1)   ->  x++
   :: (x == 2)   ->  x--
   :: skip       ->  // …do smth by default…
fi;
```

Example: Atomicity

```
byte state = 1;

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

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

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

Example: Atomicity

```
byte state = 1;

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

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

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

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

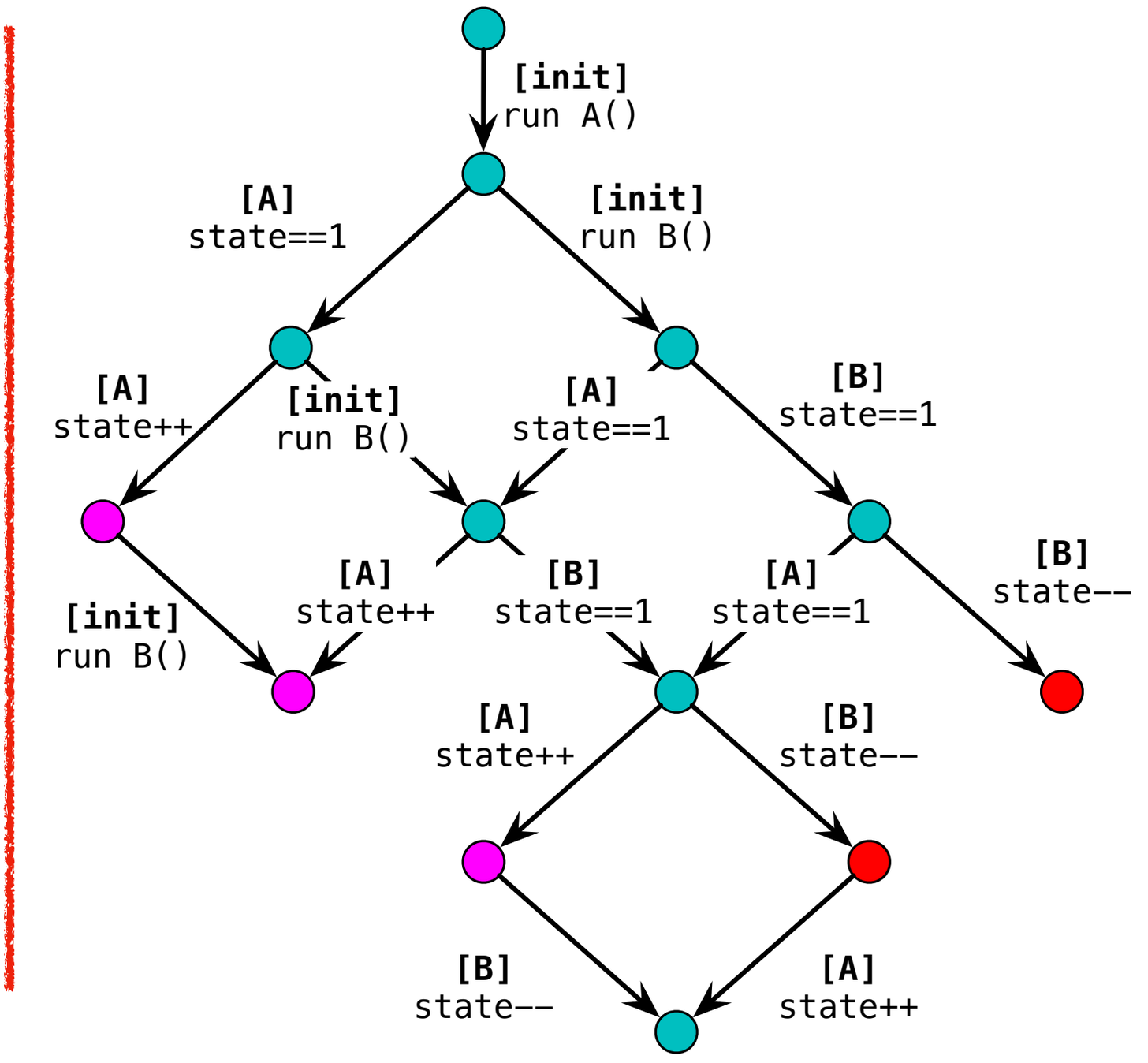
Example: Atomicity

```
byte state = 1;

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

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

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



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

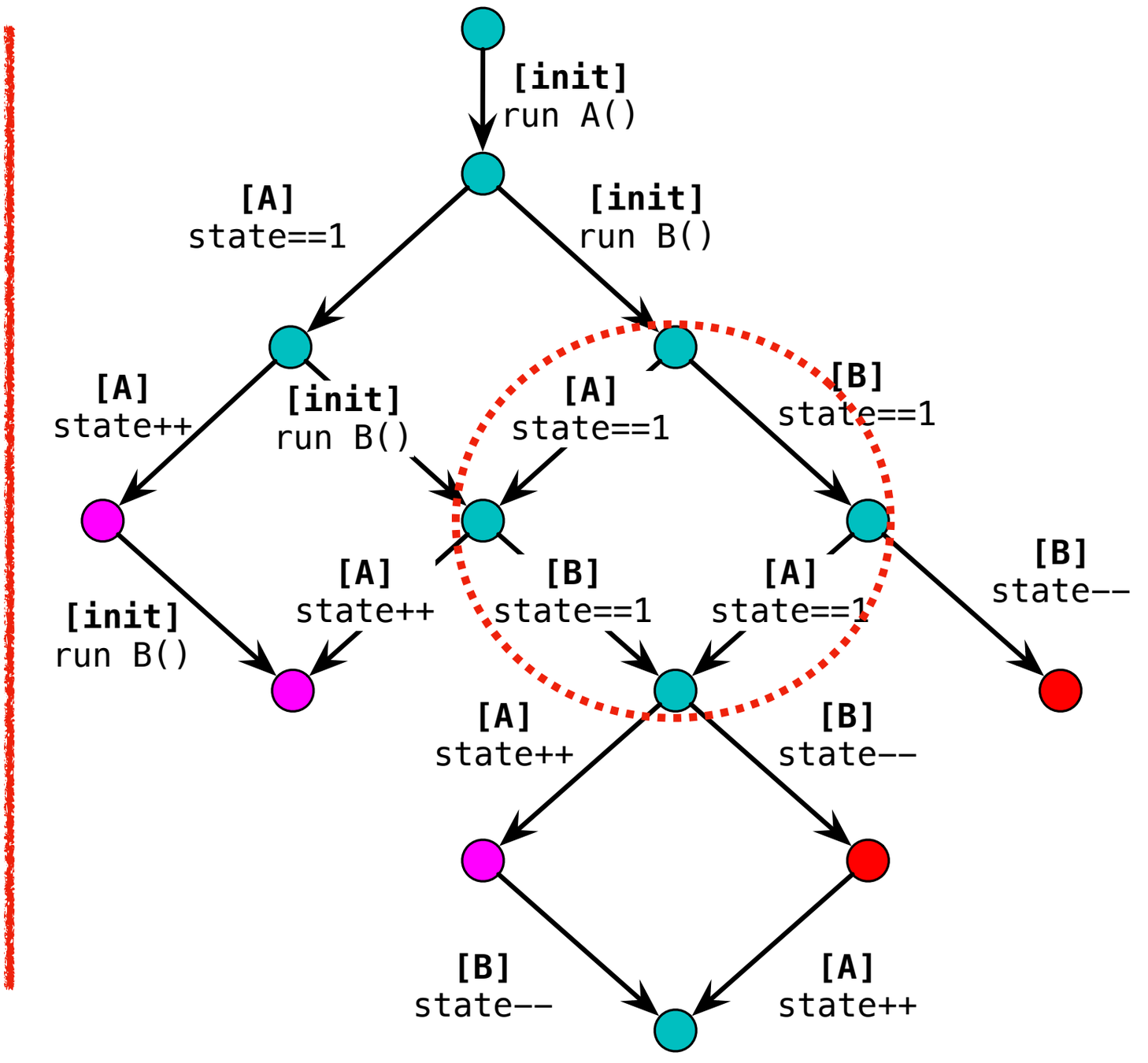
Example: Atomicity

```
byte state = 1;

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

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

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



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

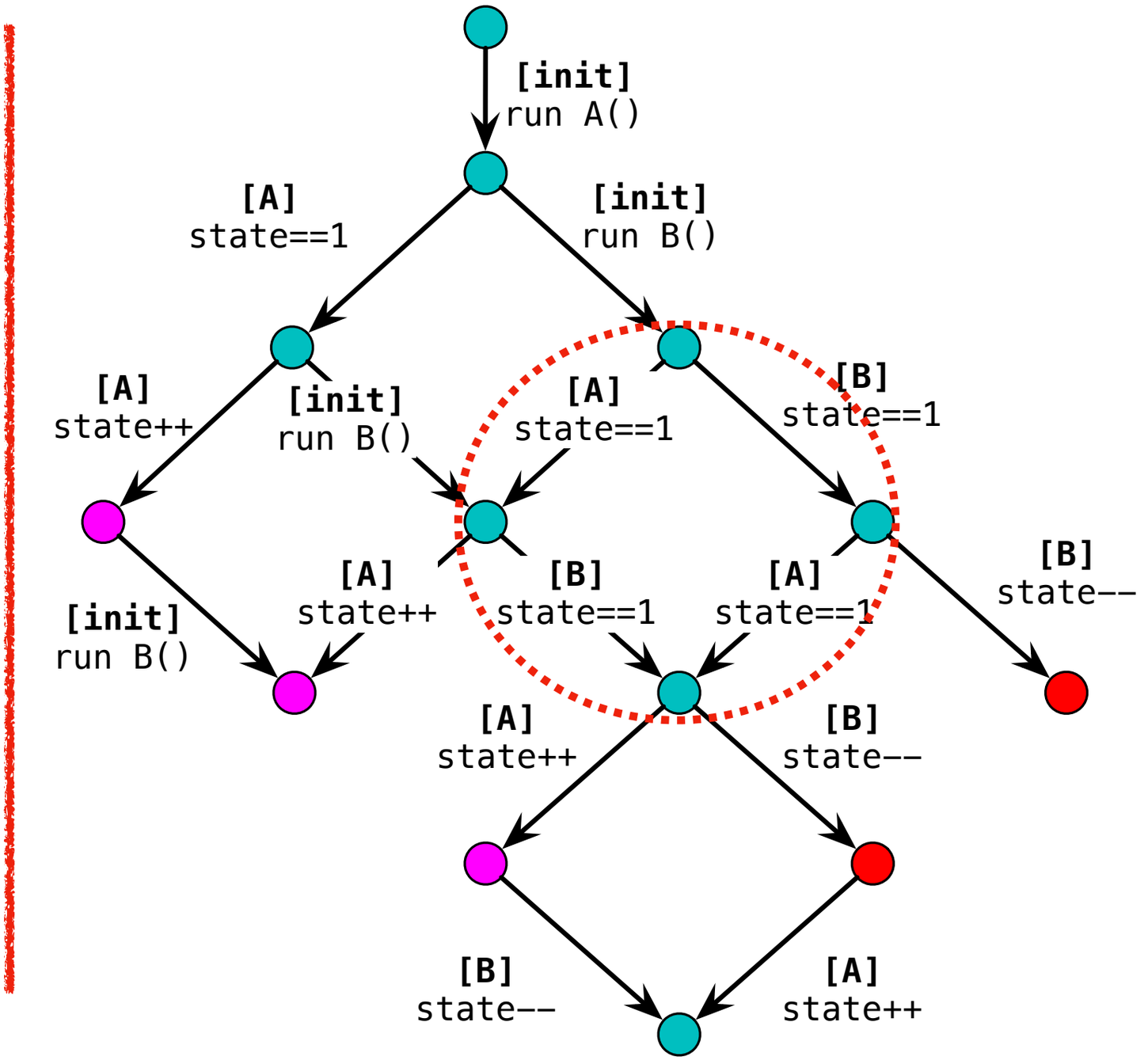
Example: Atomicity

```
byte state = 1;

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

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

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



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

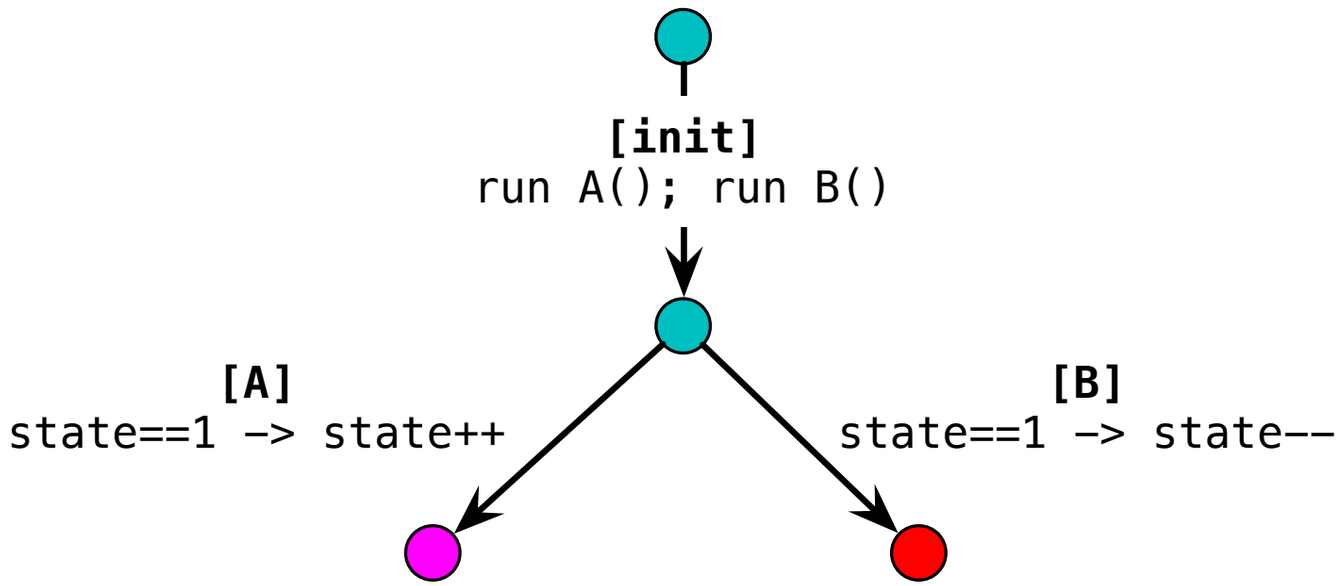
Example: Atomicity

```
byte state = 1;

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

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

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



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

`atomic`

- A process whose control is inside **`atomic`** `{ }` executes without being interrupted by other processes
- **NB!** Make sure that the sequence cannot be blocked inside. Otherwise, a process will be suspended.

Other use of `atomic`

To group complex manipulations into a single transition:

```
atomic {
    cnt = 0;

    do
      :: (cnt < max) -> z[cnt] = 3; cnt++
      :: (cnt >= max) -> break
    od
}
```

d_step

When the manipulations are deterministic and always executable, d_step is more efficient:

```
d_step {
    cnt = 0;

    do
      :: (cnt < max) -> z[cnt] = 3; cnt++
      :: (cnt >= max) -> break
    od
}
```

NB! If any statement inside d_step is blocked, you will get the run-time error

Guarded statements

timeout statement:

```
timeout -> s_0; . . .; s_n;
```

- `s_0; . . .; s_n;` execute only if no other processes are executable.
- it allows to escape deadlocks.

---

unless statement:

```
{s_0; . . .} unless {c_0 -> s_0'; . . .}
```

- `{s_0; . . .}` execute **until** `c_0` becomes true.
- `{s_0'; . . .}` execute **after** `c_0` becomes true.

Example: Shared Variable

```
mtype = {P, C};
mtype turn = P;

active proctype producer() {
    do
       :: (turn == P) ->
              printf("Produce\n");
              turn = C
    od
}

active proctype consumer() {
    do
       :: (turn == C) ->
              printf("Produce\n");
              turn = P
    od
}
```

Example: Shared Variable

```
mtype = {P, C};
mtype turn = P;

active proctype producer() {
    do
       :: (turn == P) ->
              printf("Produce\n");
              turn = C
    od
}

active proctype consumer() {
    do
       :: (turn == C) ->
              printf("Consume\n");
              turn = P
    od
}
```

```
$ spin test1.pml
      Produce
          Consume
      Produce
. . .
1 process created
```

Example: Shared Variable

```
mtype = {P, C};
mtype turn = P;

active [2] proctype producer() {
    do
       :: (turn == P) ->
              printf("Produce\n");
              turn = C
    od
}

active [2] proctype consumer() {
    do
       :: (turn == C) ->
              printf("Produce\n");
              turn = P
    od
}
```

Example: Shared Variable

```
mtype = {P, C};
mtype turn = P;

active [2] proctype producer() {
    do
       :: (turn == P) ->
              printf("Produce\n");
              turn = C
    od
}

active [2] proctype consumer() {
    do
       :: (turn == C) ->
              printf("Produce\n");
              turn = P
    od
}
```

Q: Do you see the problem?

```
$ spin -i test1.pml
Select a statement
   choice 3: proc  1 (producer:1) test1.pml:4 (state 4) [((turn==P))]
   choice 4: proc  0 (producer:1) test1.pml:4 (state 4) [((turn==P))]
Select [1-4]: 3
Select a statement
   choice 3: proc  1 (producer:1) test1.pml:6 (state 2) [printf('Produce\\n')]
   choice 4: proc  0 (producer:1) test1.pml:4 (state 4) [((turn==P))]
Select [1-4]: 3
          Produce
Select a statement
   choice 3: proc  1 (producer:1) test1.pml:7 (state 3) [turn = C]
   choice 4: proc  0 (producer:1) test1.pml:4 (state 4) [((turn==P))]
Select [1-4]: 4
Select a statement
   choice 3: proc  1 (producer:1) test1.pml:7 (state 3) [turn = C]
   choice 4: proc  0 (producer:1) test1.pml:6 (state 2) [printf('Produce\\n')]
Select [1-4]: 4
      Produce
Select a statement
   choice 3: proc  1 (producer:1) test1.pml:7 (state 3) [turn = C]
   choice 4: proc  0 (producer:1) test1.pml:7 (state 3) [turn = C]
Select [1-4]:
. . .
```

Both producers can pass the guard (**`turn == P`**)

Communication via Channels

Message Channels

- A channel is a **FIFO** message queue.
- A channel can be used to exchange messages among processes.
- Two channel types:
    - buffered channels and
    - synchronous channels (a.k.a. *rendezvous ports*).

Buffered Channels

- A channel storing up to 16 messages, each consisting of 3 fields if the listed types:

```
chan qname = [16] of {short, byte, bool}
```

Buffered Channels

- A channel storing up to 16 messages, each consisting of 3 fields if the listed types:

```
chan qname = [16] of {short, byte, bool}
```

- Useful pre-defined functions:

```
len, empty, nempty, full, nfull
```

Buffered Channels

- A channel storing up to 16 messages, each consisting of 3 fields if the listed types:

  chan qname = [16] **of** {**short, byte, bool**}

- Useful pre-defined functions:

  len, empty, nempty, full, nfull

- Send a message: qname!ex1,ex2,ex3.
  Process blocks if **qname** is full.

- Receive a message: qname?v1,v2,v3.
  Process blocks if **qname** is empty.

Buffered Channels

- A channel storing up to 16 messages, each consisting of 3 fields if the listed types:

  chan qname = [16] **of** {**short, byte, bool**}

- Useful pre-defined functions:

  len, empty, nempty, full, nfull

- Send a message: qname!ex1,ex2,ex3.
  Process blocks if **qname** is full.

- **Not-modifying** message receive: qname?[v1,v2,v3].
  Process blocks if **qname** is empty.

Buffered Channels

- A channel storing up to 16 messages, each consisting of 3 fields if the listed types:

```
chan qname = [16] of {short, byte, bool}
```

- Alternative syntax for message send:

```
qname!ex1(ex2,ex3).
```

- Alternative syntax for message recieve:

```
qname?v1(v2,v3).
```

Synchronous Channels

- A synchronous channel has size 0:

```
chan p = [0] of { byte }
```

- Messages are exchanged, not stored.

- Synchronous execution: a process executes send at the same time another process receives a message (atomic operation).

Example: Sync Channel

```
mtype = { msgtype };
chan glob = [0] of {mtype, byte};

active proctype A() {
    byte x = 124;
    printf("Send %d\n", x);
    glob!msgtype(x);

    x = 121;
    printf("Send %d\n", x);
    glob!msgtype(x);
}

active proctype B() {
    byte y;
    glob?msgtype(y);
    printf("Received %d\n", y);

    glob?msgtype(y);
    printf("Received %d\n", y);
}
```

Example: Sync Channel

```
mtype = { msgtype };
chan glob = [0] of {mtype, byte};

active proctype A() {
    byte x = 124;
    printf("Send %d\n", x);
    glob!msgtype(x);

    x = 121;
    printf("Send %d\n", x);
    glob!msgtype(x);
}

active proctype B() {
    byte y;
    glob?msgtype(y);
    printf("Received %d\n", y);

    glob?msgtype(y);
    printf("Received %d\n", y);
}
```

Example: Sync Channel

```
mtype = { msgtype };
chan glob = [0] of {mtype, byte};

active proctype A() {
    byte x = 124;
    printf("Send %d\n", x);
    glob!msgtype(x);

    x = 121;
    printf("Send %d\n", x);
    glob!msgtype(x);
}

active proctype B() {
    byte y;
    glob?msgtype(y);
    printf("Received %d\n"

    glob?msgtype(y);
    printf("Received %d\n"
}
```

```
$ spin Test1.pml
      Send 124
      Send 121
          Received 124
          Received 121
2 processes created
```

Sorted Send !!

- A message is inserted into a channel before the first message that follows it in a numerical order.
- Syntax: `chname!!value.`
- e.g.
  - `c!3; c!1;` gives `c([3, 1]).`
  - `c!!3; c!!1;` gives `c([1, 3]).`

Random Receive ??

- Executable if there is at least one message the channel that can be received, **regardless of its position.**
- Syntax: `chname??value.`
- given `c([3, 1])`
    - `c?1` is blocked since `1` is not accessible but
    - `c??1` will work.

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

(1,2) (1,1) (1,3) (0,1)

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

(1,1) (0,1)

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

(0,1) (1,1) (1,2) (1,3)

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

Example: Senders and Receivers

```
proctype S1() {
    c!1,2;   c!1,1;
    c!1,3;   c!0,1;
}

proctype R1() {
    do
      :: c?v1,v2 ->
         printf(«(%d,%d)\n", v1, v2)
    od
}
```

```
proctype S2() {
    c!!1,2;   c!!1,1;
    c!!1,3;   c!!0,1;
}

proctype R2() {
    do
      :: c??v1,1 ->
         printf("(%d,%d)\n", v1, 1)
    od
}
```

Q: What will the receiver print?

(0,1) (1,1)

Labels

End-state labels

- are used to mark **valid end-states** and tell them apart from deadlocks,
- includes any label starting with ‘**`end`**’.

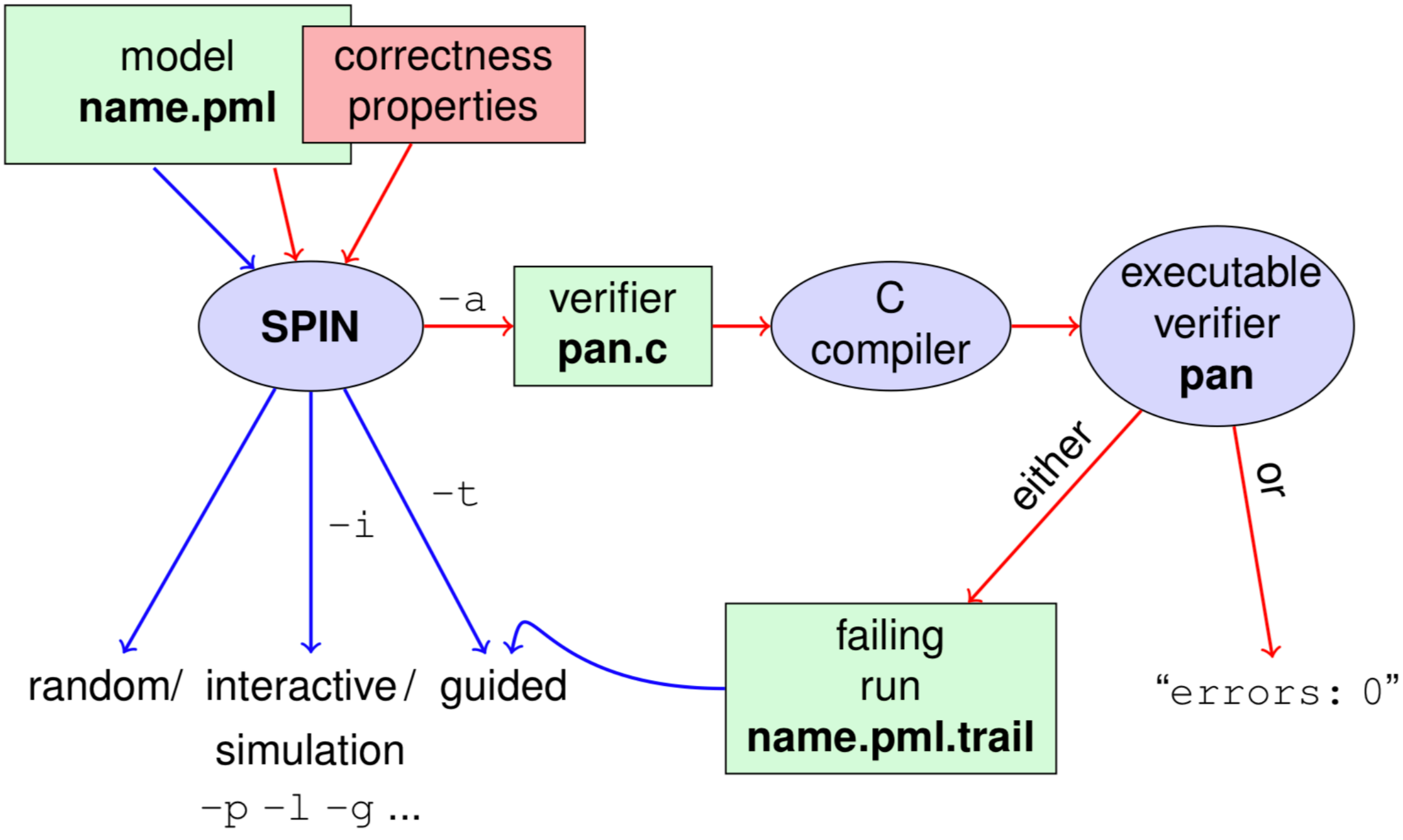
By **default**, the only valid end-state is reached when the process reaches the *syntactic end* of its body.

Progress-state Labels

- are used to mark a state that must be executed for a process to make progress (~liveness),
- includes any label starting with '**`progress`**'.

Any infinite cycle that does not cross a progress state is a potential starvation loop.

Basic Verification


model
name.pml
correctness
properties
SPIN
-a
verifier
pan.c
C
compiler
executable
verifier
pan
either
or
-t
-i
random/ interactive / guided
simulation
-p -l -g ...
failing
run
name.pml.trail
"errors: 0"

Assertions

```
chan com = [0] of {byte};
byte var;

proctype p() {
    byte i;
    do
        :: if
               :: i >= 5 -> break;
               :: else -> printf(«Doing something else\n»); i++
           fi
        :: com ? var -> printf(«p received %d\n», var)
    od
}

init {
    run p();
    end: com ! 100;
}
```

Assertions

```
chan com = [0] of {byte};
byte var;

proctype p() {
    byte i;
    do
        :: if
               :: i >= 5 -> break;
               :: else -> printf(«Doing something else\n»); i++
           fi
        :: com ? var -> printf(«p received %d\n», var)
    od
}

init {
    run p();
    end: com ! 100;
}
```

Q: Is it possible that **p** does not read from **com** at all?

Assertions

```
chan com = [0] of {byte};
byte var;

proctype p() {
    byte i;
    do
        :: if
               :: i >= 5 -> break;
               :: else -> printf(«Doing something else\n»); i++
           fi
        :: com ? var -> printf(«p received %d\n», var)
    od
}

init {
    run p();
    end: com ! 100; // end is not a deadlock
}
```

Q: Is it possible that **p** does not read from **com** at all? **YES**

```
$ spin -i Test1.pml
Select a statement
   choice 1: proc  1 (p:1) Test1.pml:5 (state 10) [IF]
   choice 3: proc  0 (:init::1) Test1.pml:16 (state 2) <valid end state> [com!100]
Select [1-3]: 1
Select stmnt (proc  1 (p:1) )
   choice 0: other process
Select [0-2]: 2
Select a statement
   choice 1: proc  1 (p:1) Test1.pml:8 (state 4) [printf('Doing something else\\n')]
Select [1-2]: 1
          Doing something else
Select a statement
   choice 1: proc  1 (p:1) Test1.pml:8 (state 5) [i = (i+1)]
Select [1-2]: 1
. . .
   choice 1: proc  1 (p:1) Test1.pml:8 (state 4) [printf('Doing something else\\n')]
Select [1-2]: 1
          Doing something else
. . .
          Doing something else
. . .
          Doing something else
. . .
          Doing something else
. . .
Select stmnt (proc  1 (p:1) )
   choice 0: other process
   choice 1: ((i>=5))
Select [0-2]: 1
Select a statement
   choice 1: proc  1 (p:1) Test1.pml:5 (state 12) [break]
Select [1-2]: 1
. . .
#processes: 1
      var = 0
 18:  proc  0 (:init::1) Test1.pml:16 (state 2) <valid end state>
2 processes created
```

interactive simulation

Assertions

```
chan com = [0] of {byte};
byte var;

proctype p() {
    byte i;
    do
        :: if
               :: i >= 5 -> break;
               :: else -> printf(«Doing something else\n»); i++
           fi
        :: com ? var -> printf(«p received %d\n», var)
    od
    assert(var == 100);  // check the value of var
}

init {
    run p();
    end: com ! 100; // end is not a deadlock
}
```

Q: Is it possible that **p** does not read from **com** at all? **YES**

```
$ spin -a Test1.pml && gcc -o pan pan.c && ./pan
pan:1: assertion violated (var==100) (at depth 12)
pan: wrote Test1.pml.trail

(Spin Version 6.4.9 -- 17 December 2018)
Warning: Search not completed
  + Partial Order Reduction

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

State-vector 36 byte, depth reached 12, errors: 1
       13 states, stored
        0 states, matched
       13 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.001 equivalent memory usage for states (stored*(State-vector + overhead))
    0.291 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage

pan: elapsed time 0 seconds
```

checks assertions

indicates sequence of steps leading to errors

```
$ spin -t -p Test1.pml
Starting p with pid 1
  1: proc  0 (:init::1) Test1.pml:16 (state 1)[(run p())]
  2: proc  1 (p:1) Test1.pml:8 (state 3)[else]
          Doing something else
  3: proc  1 (p:1) Test1.pml:8 (state 4)[printf('Doing something else\\n')]
  3: proc  1 (p:1) Test1.pml:8 (state 5)[i = (i+1)]
  4: proc  1 (p:1) Test1.pml:8 (state 3)[else]
          Doing something else
  5: proc  1 (p:1) Test1.pml:8 (state 4)[printf('Doing something else\\n')]
  5: proc  1 (p:1) Test1.pml:8 (state 5)[i = (i+1)]
  6: proc  1 (p:1) Test1.pml:8 (state 3)[else]
          Doing something else
  7: proc  1 (p:1) Test1.pml:8 (state 4)[printf('Doing something else\\n')]
  7: proc  1 (p:1) Test1.pml:8 (state 5)[i = (i+1)]
  8: proc  1 (p:1) Test1.pml:8 (state 3)[else]
          Doing something else
  9: proc  1 (p:1) Test1.pml:8 (state 4)[printf('Doing something else\\n')]
  9: proc  1 (p:1) Test1.pml:8 (state 5)[i = (i+1)]
 10: proc  1 (p:1) Test1.pml:8 (state 3)[else]
          Doing something else
 11: proc  1 (p:1) Test1.pml:8 (state 4)[printf('Doing something else\\n')]
 11: proc  1 (p:1) Test1.pml:8 (state 5)[i = (i+1)]
 12: proc  1 (p:1) Test1.pml:7 (state 1)[((i>=5))]
spin: Test1.pml:12, Error: assertion violated
spin: text of failed assertion: assert((var==100))
 13: proc  1 (p:1) Test1.pml:12 (state 13)[assert((var==100))]
spin: trail ends after 13 steps
#processes: 2
     var = 0
 13: proc  1 (p:1) Test1.pml:13 (state 14) <valid end state>
 13: proc  0 (:init::1) Test1.pml:17 (state 2) <valid end state>
2 processes created
```

replays the error-trail

Time for exercises…