CS477 Formal Software Development Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides mostly a reproduction of Theo C. Ruys – SPIN Beginners' Tutorial

April 11, 2018

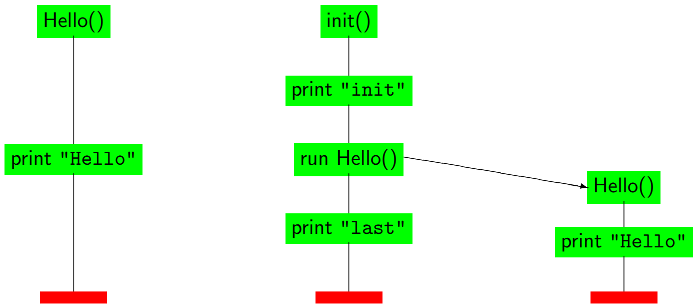
Hello World

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
printf("Hello process, my pid is: %d\n", _pid);
}
init {
      int lastpid;
      printf("init process, my pid is: %d\n", _pid);
      lastpid = run Hello();
      printf("last pid was: %d\n", lastpid);
}
```

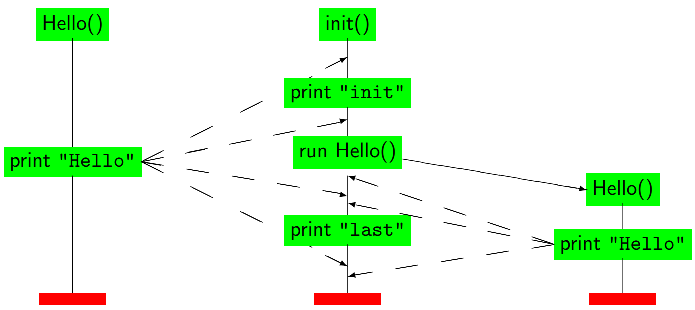

Hello World, Sample Execution

```
bash-3.2$ spin hello.pml
          init process, my pid is: 1
      Hello process, my pid is: 0
              Hello process, my pid is: 2
          last pid was: 2
3 processes created
bash-3.2$ spin hello.pml
      Hello process, my pid is: 0
          init process, my pid is: 1
          last pid was: 2
              Hello process, my pid is: 2
3 processes created
```


Hello Processes



Hello Processes Interleavings



Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved
  - Only one process can execute a statement at each point in time.
  - Exception: rendez-vous communication.
- All statements are atomic
  - Each statement is executed without interleaving it parts with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - Only one choice is made, non-deterministically (randomly).

Variables and Types (1)

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.
- mtype (message type) one user-defined enum type

Basic types

```
bit   turn=1;     [0..1]
bool  flag;       [0..1]
byte  counter;    [0..255]
short s;          [-2^16-1.. 2^16 –1]
int   msg;        [-2^32-1.. 2^32 –1]
```

Arrays — array indicing start at 0

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

Typedef (records)

```
typedef Record {
  short f1;
  byte  f2;
}
Record rr;
rr.f1 = ..
```

variable declaration

Variables and Types (2)

- Variables should be declared.
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing (see communication)
- Variables can be used in expressions.

Most arithmetic, relational, and logical operators of C/Java are supported, including bitshift operators.

```
int ii;
bit bb;

bb=1;
ii=2;

short s=-1;

typedef Foo {
  bit bb;
  int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;

ii*s+27 == 23;
printf("value: %d", s*s);
```

assignment =

declaration + initialisation

equal test ==

Statements (1)

- The body of a process consists of a sequence of statements. A statement is either
  - executable: the statement can be executed immediately.
  - blocked: the statement cannot be executed.
- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero.

executable/blocked depends on the global state of the system.

`2 < 3`	always executable
`x < 27`	only executable if value of `x` is smaller `27`
`3 + x`	executable if `x` is not equal to `–3`

Statements (2)

Statements are separated by a semi-colon: ";".

- The **skip** statement is always executable.
  - "does nothing", only changes process' process counter
- 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 not evaluated during verification, of course).

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

Executable if `Noot` can be created…

Can only become executable if a some other process makes `x` greater than 2.

Statements (3)

- **assert(<expr>);**
  - The **assert**-statement is always executable.
  - If **<expr>** evaluates to zero, SPIN will exit with an error, as the **<expr>** "has been violated".
  - The **assert**-statement is often used within Promela models, to check whether certain properties are valid in a state.

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

proctype receiver() {
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```


DEMO

Mutual Exclusion (1)

WRONG!

```
bit  flag;   /* signal entering/leaving the section */
byte mutex;  /* # procs in the critical section.    */

proctype P(bit i) {
  flag != 1;
  flag  = 1;
  mutex++;
  printf("MSC: P(%d) has entered section.\n", i);
  mutex--;
  flag  = 0;
}

proctype monitor() {
  assert(mutex != 2);
}

init {
  atomic { run P(0); run P(1); run monitor(); }
}
```

models:
`while (flag == 1) /* wait */;`

Problem: assertion violation! Both processes can pass the `flag != 1` "at the same time", i.e. before `flag` is set to `1`.

starts two instances of process P

DEMO

WRONG!

Mutual Exclusion (2)

```
bit  x, y;    /* signal entering/leaving the section */
byte mutex;   /* # of procs in the critical section.  */

active proctype A() {              active proctype B() {
  x = 1;                             y = 1;
  y == 0;                            x == 0;
  mutex++;                           mutex++;
  mutex--;                           mutex--;
  x = 0;                             y = 0;
}                                  }

active proctype monitor() {
  assert(mutex != 2);
}
```

Process A waits for process B to end.

Problem: invalid-end-state! Both processes can pass execute `x = 1` and `y = 1` "at the same time", and will then be waiting for each other.

DEMO

Dekker [1962]

Mutual Exclusion (3)

```
bit  x, y;    /* signal entering/leaving the section  */
byte mutex;   /* # of procs in the critical section.  */
byte turn;    /* who's turn is it?                     */

active proctype A() {           active proctype B() {
  x = 1;                          y = 1;
  turn = B_TURN;                  turn = A_TURN;
  y == 0 ||                       x == 0 ||
    (turn == A_TURN);               (turn == B_TURN);
  mutex++;                        mutex++;
  mutex--;                        mutex--;
  x = 0;                          y = 0;
}                               }

active proctype monitor() {
  assert(mutex != 2);
}
```

Can be generalised to a single process.

First "software-only" solution to the mutex problem (for two processes).

DEMO

Bakery

Mutual Exclusion (4)

```
byte turn[2];  /* who's turn is it?         */
byte mutex;    /* # procs in critical section */

proctype P(bit i) {
  do
  :: turn[i] = 1;
     turn[i] = turn[1-i] + 1;
     (turn[1-i] == 0) || (turn[i] < turn[1-i]);
     mutex++;
     mutex--;
     turn[i] = 0;
  od
}

proctype monitor() { assert(mutex != 2); }
init { atomic {run P(0); run P(1); run monitor()}}
```

Problem (in Promela/SPIN): `turn[i]` will overrun after 255.

More mutual exclusion algorithms in (good-old) [Ben-Ari 1990].

if-statement (1)

inspired by: Dijkstra's guarded command language

```
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) executable, the **if**-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no **choice**$_i$ is executable, the **if**-statement is blocked.
- The operator "**->**" is equivalent to "**;**". By convention, it is used within **if**-statements to separate the guards from the statements that follow the guards.

if-statement (2)

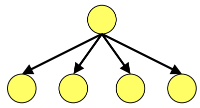
```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)     -> n=n-2
:: (n % 3 == 0) -> n=3
:: else         -> skip
fi
```

- The **else** guard becomes executable if none of the other guards is executable.

give n a random value

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

non-deterministic branching



skips are redundant, because assignments are themselves always executable...

do-statement (1)

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

- With respect to the choices, a **do**-statement behaves in the same way as an **if**-statement.
- However, instead of ending the statement at the end of the choosen list of statements, a **do**-statement repeats the choice selection.
- The (always executable) **break** statement exits a **do**-loop statement and transfers control to the end of the loop.

do-statement (2)

- Example – modelling a traffic light

if- and do-statements are ordinary Promela statements; so they can be nested.

```
mtype = { RED, YELLOW, GREEN } ;

active proctype TrafficLight() {
    byte state = GREEN;
    do
    ::  (state == GREEN)  -> state = YELLOW;
    ::  (state == YELLOW) -> state = RED;
    ::  (state == RED)    -> state = GREEN;
    od;
}
```

mtype (message type) models enumerations in Promela

Note: this do-loop does not contain any non-deterministic choice.

Communication

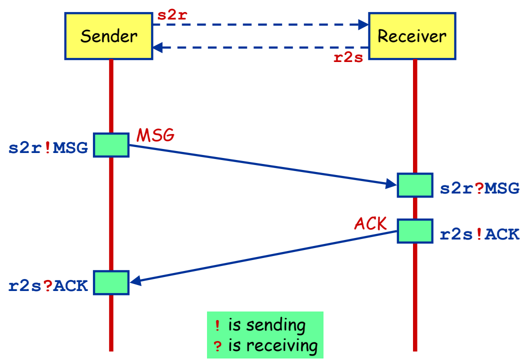
Major models of communication

1. Shared variables
   - one writes, many read later
2. Point-to-Point synchronous message passing
   - one sends, one other receives at the same time
   - send blocks until recieve can happen
3. Point-to-Point asynchronous message passing
   - one sends, one other receives some time later
   - send never blocks
4. Point-to-Point buffered message passing
   - When buffer not full behaves like asynchronous
   - When buffer full, two variations: block or drop message
   - send never blocks
5. Synchronous broadcast
   - one sends, many receive synchronously
   - First variation: send never blocks process may receive if ready to ready
   - Second variation: send blocks until all possible recipients ready to receive

Communication in SPIN

- With more or less complexity each can implement the others
- Spin supports 1 and 4 (blocks send when buffer full), but with bounded buffers
- Buffer size = 0 $\Longrightarrow$ synchronous communication
- Large buffer size approximates asynchronous communication

Communication (1)



Communication (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:

also called: queue or buffer

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

name of the channel

type of the elements that will be transmitted over the channel

number of elements in the channel
dim==0 is special case: rendez-vous

```
chan c        = [1] of {bit};
chan toR      = [2] of {mtype, bit};
chan line[2]  = [1] of {mtype, Record};
```

array of channels

Communication (3)

- channel = FIFO-buffer (for dim>0)

! Sending - *putting a message into a channel*

```
ch ! <expr1>, <expr2>, … <exprn>;
```

- The values of <$expr_i$> should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

? Receiving - *getting a message out of a channel*

<var> + <const> can be mixed

```
ch ? <var1>, <var2>, … <varn>;
```

message passing

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the <var_i>s.

```
ch ? <const1>, <const2>,  … <constn>;
```

message testing

- If the channel is not empty and the message at the front of the channel evaluates to the individual <$const_i$>, the statement is executable and the message is removed from the channel.

Communication (4)

- Rendez-vous communication
  `<dim> == 0`
  The number of elements in the channel is now zero.
  - If send `ch!` is enabled and if there is a corresponding receive `ch?` that can be executed simultaneously and the constants match, then both statements are enabled.
  - Both statements will "handshake" and together take the transition.
- *Example:*
  `chan ch = [0] of {bit, byte};`
  - P wants to do `ch ! 1, 3+7`
  - Q wants to do `ch ? 1, x`
  - Then after the communication, **x** will have the value `10`.

DEMO

Alternating Bit Protocol (1)

- Alternating Bit Protocol
  - To every message, the sender adds a bit.
  - The receiver acknowledges each message by sending the received bit back.
  - To receiver only excepts messages with a bit that it excepted to receive.
  - If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

DEMO

Alternating Bit Protocol (2)

```
mtype {MSG, ACK};

chan toS = [2] of {mtype, bit};
chan toR = [2] of {mtype, bit};

proctype Sender(chan in, out)
{
  bit sendbit, recvbit;
  do
  :: out ! MSG, sendbit ->
       in ? ACK, recvbit;
       if
       :: recvbit == sendbit ->
          sendbit = 1-sendbit
       :: else
       fi
  od
}
```

channel length of 2

```
proctype Receiver(chan in, out)
{
  bit recvbit;
  do
  :: in ? MSG(recvbit) ->
     out ! ACK(recvbit);
  od
}

init
{
  run Sender(toS, toR);
  run Receiver(toR, toS);
}
```

Alternative notation:
`ch ! MSG(par1, …)`
`ch ? MSG(par1, …)`

atomic

atomic { stat$_1$; stat$_2$; ... stat$_n$ }

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if **stat$_1$** is executable — no pure atomicity
- if a **stat$_i$** (with **i>1**) is blocked, the "atomicity token" is (temporarily) lost and other processes may do a step

- (Hardware) solution to the mutual exclusion problem:

```
proctype P(bit i) {
  atomic {flag != 1; flag = 1; }
  mutex++;
  mutex--;
  flag  = 0;
}
```


d_step

d_step { stat$_1$; stat$_2$; ... stat$_n$ }

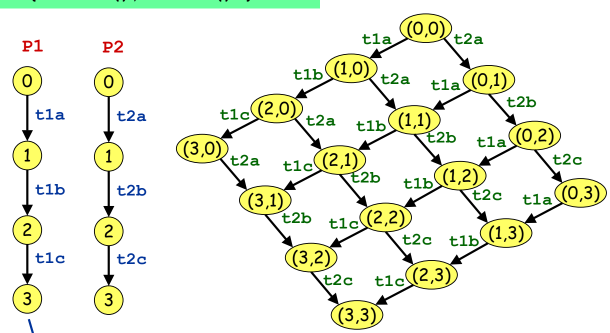
- more efficient version of **atomic**: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if **stat$_i$ (i>1)** blocks.
- **d_step** is especially useful to perform intermediate computations in a single transition

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

- **atomic** and **d_step** can be used to lower the number of states of the model

No atomicity

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

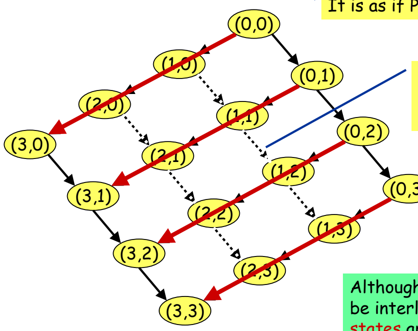


Not completely correct as each process has an implicit end-transition…

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

atomic

It is as if P1 has only one transition…



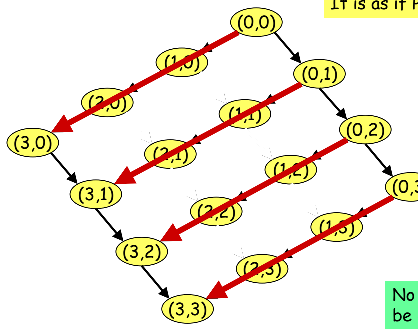
If one of P1's transitions blocks, these transitions may get executed

Although atomic clauses cannot be interleaved, the intermediate states are still constructed.


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

d_step

It is as if P1 has only one transition…



No intermediate states will be constructed.

Checking for pure atomicity

- Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:

```
bit aflag;
```

2. Change all atomic clauses to:

```
atomic {
  stat1;
  aflag=1;
  stat2
  ...
  statn
  aflag=0;
}
```

3. Check that `aflag` is always 0.

```
[]!aflag
```

e.g.

```
active process monitor {
  assert(!aflag);
}
```


timeout (1)

- Promela does not have real-time features.
  - In Promela we can only specify functional behaviour.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.
- `timeout`
  - SPIN's `timeout` becomes executable if there is no other process in the system which is executable
  - so, `timeout` models a global timeout
  - `timeout` provides an escape from deadlock states
  - beware of statements that are always executable…

timeout (1)

- Promela does not have real-time features.
  - In Promela we can only specify functional behaviour.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.
- `timeout`
  - SPIN's `timeout` becomes executable if there is no other process in the system which is executable
  - so, `timeout` models a global timeout
  - `timeout` provides an escape from deadlock states
  - beware of statements that are always executable…

goto

`goto label`

- transfers execution to `label`
- each Promela statement might be labelled
- quite useful in modelling communication protocols

```
wait_ack:
  if
  :: B?ACK -> ab=1-ab ; goto success
  :: ChunkTimeout?SHAKE ->
     if
     :: (rc < MAX)  -> rc++; F!(i==1),(i==n),ab,d[i];
                       goto wait_ack
     :: (rc >= MAX) -> goto error
     fi
  fi ;
```

Timeout modelled by a channel.

Part of model of BRP

unless

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

- Statements in *<stats>* are executed until the first statement (*guard*) in the escape sequence becomes executable.
- resembles exception handling in languages like Java
- *Example:*

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


unless

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

- Statements in *<stats>* are executed until the first statement (*guard*) in the escape sequence becomes executable.
- resembles exception handling in languages like Java
- *Example:*

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

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 57 University of Twente
Elsa L Gunter CS477 Formal Software Development Method / 39

inline - poor man's procedures

- Promela also has its own macro-expansion feature using the **inline**-construct.

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

Should be declared somewhere else (probably as a local variable). [annotation on `i=0;`]

Be sure to reset temporary variables. [annotation on final `i=0;`]

  - error messages are more useful than when using **#define**
  - cannot be used as expression
  - all variables should be declared somewhere else