Spin: Overview of PROMELA*

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/trentin

Formal Methods Lab Class, March 02, 2018

Università degli Studi di Trento

(compiled on 17/05/2018 at 12:50)

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/18

Contents

PROMELA

PROMELA is not a programming language,
but rather a **meta-language for** building **verification models**.

- The design of PROMELA is focused on the interaction among processes at the system level;
- Provides:
  - non-deterministic control structures,
  - primitives for process creation,
  - primitives for interprocess communication.
- Misses:
  - functions with return values,
  - expressions with side-effects,
  - data and functions pointers.

Types of objects

Three basic types of objects:

- processes
- data objects
- message channels

+ labels

Contents

Process Initialization [1/3]

- **active**: process created at initialization phase

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

Process Initialization [1/3]

- **active**: process created at initialization phase

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

- $init$ is a process that is *active* in the initial system state.

  $\Longrightarrow$ commonly used to initialize system

Process Initialization [1/3]

- **active**: process created at initialization phase

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

- init is a process that is *active* in the initial system state.

  $\Longrightarrow$ commonly used to initialize system

- `init` + `active` processes $\Longrightarrow$ instantiated in declaration order

Process Initialization [1/3]

- **active**: process created at initialization phase

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

- $init$ is a process that is *active* in the initial system state.

  $\Longrightarrow$ commonly used to initialize system

- `init` + `active` processes $\Longrightarrow$ instantiated in declaration order
- **run**: process created when instruction is processed

```
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);
}
```

note: run allows for input parameters!

Process Initialization [2/3]

- No parameter can be given to *init* nor to active processes.

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

- ```
  ~$ spin test.pml
        x = 0
  ```
  ```

Process Initialization [2/3]

- No parameter can be given to *init* nor to active processes.

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

- ```
  ~$ spin test.pml
        x = 0
  ```

All parameters of an active process default to 0.
  ```

Process Initialization [2/3]

- No parameter can be given to *init* nor to active processes.

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

- ```
  ~$ spin test.pml
        x = 0
  ```

All parameters of an active process default to 0.

- A process does not necessarily start right after creation

```
proctype proc (byte x) {
    printf("x = %d\n", x);
}
init {
    run proc(0);
    run proc(1);
}
```

- ```
  ~$ spin test.pml
        x = 0
            x = 1
  ```
- ```
  ~$ spin test.pml
            x = 1
        x = 0
  ```
  ```

Process Initialization [3/3]

- Only a limited number of processes (up to 255) can be created:

```
proctype proc(byte x) {
    printf("x = %d\n", x);
    run proc(x + 1)
}
init {
    run proc(0);
}
```

- ```
  ~$ spin test.pml
      x = 0
          x = 1
              x = 2
                  ...
  spin: too many processes (255 max)
  timeout
  ```
  ```

Process Initialization [3/3]

- Only a limited number of processes (up to 255) can be created:

```
proctype proc(byte x) {
    printf("x = %d\n", x);
    run proc(x + 1)
}
init {
    run proc(0);
}
```

- ```
  ~$ spin test.pml
        x = 0
            x = 1
                x = 2
                    ...
  spin: too many processes (255 max)
  timeout
  ```

- A process "terminates" when it reaches the end of its code.
- A process "dies" when it has terminated and all processes created after it have died.
  ```

Process Execution [1/2]

- Processes execute **concurrently** with all other processes.
- Processes are scheduled **non-deterministically**.
- Processes are **interleaved**: statements of different processes do not occur at the same time (except for synchronous channels).
- Each process may have several different possible actions enabled at each point of execution: only one choice is made (non-deterministically).

Process Execution [2/2]

- Each process has its own local state:
  - process id **_pid**;
  - value of the local variables.

- A process communicates with other processes:
  - using global (shared) variables (might need synchronization!);
  - using channels.

Statements [1/6]

- each statement is **atomic**
- Every statement is either *executable* or *blocked*.

Statements [1/6]

- each statement is **atomic**
- Every statement is either *executable* or *blocked*.

- Always executable:
  - print statements
  - assignments
  - skip
  - assert
  - break
  - ...

Statements [1/6]

- each statement is **atomic**
- Every statement is either *executable* or *blocked*.

- Always executable:
  - print statements
  - assignments
  - skip
  - assert
  - break
  - ...

- Not always executable:
  - the **run** statement is executable only if there are less than 255 processes alive;
  - **timeout**: executable only when there is **no other executable process**
  - expressions

Statements [2/6]

- 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;

- **Busy-Waiting:** the expression `(a == b);` is equivalent to:
  `while (a != b) { skip }; /* C-code */`

- Expressions must be side-effect free
  (e.g. `b = c++` is not valid).

Statements [3/6]

selection:

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

repetition:

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

- { `s_i; ...` } executed only if `c_i` is executable
- if more than one `c_i` is excutable, then executed branch is chosen **non-deterministically**
- if no `c_i` is executable, then **else** branch is executed –if present
- **break**: exit from loop

Statements [4/6]

timeout

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

- `{ s_0; ... s_n; }` executed **only if** no other process is executable
- statement that acts as a *global timeout*
- allows to escape **deadlocks**

Statements [4/6]

timeout

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

- `{ s_0; ... s_n; }` executed **only if** no other process is executable
- statement that acts as a *global timeout*
- allows to escape **deadlocks**

unless

```
{ s_0; ... s_n; } unless { c_0; s_0'; ... s_n'; }
```

- `{ s_0; ... s_n; }` executed **until** c_0 becomes executable
- `{ s_0'; ... s_n'; }` executed **after** c_0 becomes executable
- similar to *exception handling*

Statements [5/6]

for

```
int i; int a[10];
for (i : 1 .. N) {
    ...
}
for (i in a) { // + channels
    ...
}
```

- also on *arrays*, e.g. `int a[10]`
- also on *channels* (peek read!), e.g. `typedef m { ... };` `chan c = [9] of { m };`

Statements [5/6]

for

```
int i; int a[10];
for (i : 1 .. N) {
    ...
}
for (i in a) { // + channels
    ...
}
```

- also on *arrays*, e.g. `int a[10]`
- also on *channels* (peek read!), e.g. `typedef m { ... };` `chan c = [9] of { m };`

select

```
select(i: 8..17);
```

- assigns `i` with a random value in the interval 8..17, bounds included

Statements [5/6]

for

```
int i; int a[10];
for (i : 1 .. N) {
    ...
}
for (i in a) { // + channels
    ...
}
```

- also on *arrays*, e.g. `int a[10]`
- also on *channels* (peek read!), e.g. `typedef m { ... };` `chan c = [9] of { m };`

select

```
select(i: 8..17);
```

- assigns `i` with a random value in the interval 8..17, bounds included

conditional expression

```
( c_0 -> e_1 : e_2 )
```

- evaluates to e_1 if c_0 is true
- evaluates to e_2 if c_0 is false

Statements [6/6]

atomic / d_step

both can be used to **group** statements in an **atomic sequence**, which are then executed *in a single step*.

```
atomic { s_0; ... s_i; ... s_n; }
```

- executable if `s_0` is executable
- temporary loss of atomicity if `s_i`, $i > 0$, not executable

```
d_step { s_0; ... s_i; ... s_n; }
```

- executable if `s_0` is executable
- run-time error if `s_i`, $i > 0$, not executable
- can only contain **deterministic** steps
- no *intermediate state* is generated

Contents

Basic types

Type	Typical 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` format specifier;
- There are no floats and no strings;

Typical declarations

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

- All variables are initialized by default to 0.
- Array indexes starts at 0.
- $\Longrightarrow$ **unique initial state** for all execution traces of one model!

Data structures

- A **run** statement accepts a list of variables or structures, but no array.
- **Simulation-only trick:** enclose array inside data structure

```
typedef Record {
    byte a[3];
    int x;
};
proctype run_me(Record r) {
    r.x = 12
}
init {
    Record test;
    run run_me(test)
}
```

- Multi-dimensional arrays are not supported, although there are indirect ways:

```
typedef Array {
        byte el[4]
};
Array a[4];
```

Variable Scope

- Spin (old versions): only two levels of scope
  - **global scope**: declaration outside all process bodies.
  - **local scope**: declaration within a process body.

Variable Scope

- Spin (old versions): only two levels of scope
  - **global scope**: declaration outside all process bodies.
  - **local scope**: declaration within a process body.
- Spin (versions 6+): added block-level scope

```
init {
    int x;
    {       /* y declared in nested block */
        int y;
        printf("x = %d, y = %d\n", x, y);
        x++;
        y++;
    }
    /* Spin Version 6 (or newer): y is not in scope,
    /* Older: y remains in scope */
    printf("x = %d, y = %d\n", x, y);
}
```

Note: since Spin version 2.0, variable declarations are not implicitly moved to the beginning of a block

Contents

Message Channels

- A channel is a FIFO (first-in first-out) message queue.
- A channel can be used to exchange messages among processes.
- Two types:
  - buffered channels,
  - synchronous channels (aka rendezvous ports)

Buffered Channels

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

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

Buffered Channels

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

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

- A message can contain any pre-defined or user-defined type.
  Note: array must be enclosed within user-defined types.

Buffered Channels

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

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

- A message can contain any pre-defined or user-defined type.
  Note: array must be enclosed within user-defined types.
- Useful pre-defined functions: `len`, `empty`, `nempty`, `full`, `nfull`:

  $\Longrightarrow$ `num_msgs_in_queue = len(qname);`

Buffered Channels

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

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

- A message can contain any pre-defined or user-defined type.
  Note: array must be enclosed within user-defined types.

- Useful pre-defined functions: `len`, `empty`, `nempty`, `full`, `nfull`:

  $\Longrightarrow$ `num_msgs_in_queue = len(qname);`

- **Message Send**:

  `qname!expr1,expr2,expr3`

  The process blocks if the channel is full.

Buffered Channels

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

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

- A message can contain any pre-defined or user-defined type.
  Note: array must be enclosed within user-defined types.
- Useful pre-defined functions: `len`, `empty`, `nempty`, `full`, `nfull`:

  $\Longrightarrow$ `num_msgs_in_queue = len(qname);`

- **Message Send**:

  ```
  qname!expr1,expr2,expr3
  ```

  The process blocks if the channel is full.
- **Message Receive**:

  ```
  qname?var1,var2,var3
  ```

  The process blocks if the channel is empty.

Alternative use of Buffered Channels

- An alternative syntax for message send/receive involves brackets:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

$\Longrightarrow$ used to highlight the first field, **e.g.** when it acts as 'message type'

Alternative use of Buffered Channels

- An alternative syntax for message send/receive involves brackets:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

  $\Longrightarrow$ used to highlight the first field, **e.g.** when it acts as 'message type'

- If - at the receiving side - some parameter is set to a constant value:

```
qname?const1,var2,var3
```

  then the process blocks if the channel is empty or the input message field does not match the fixed constant value.

  $\Longrightarrow$ used to filter messages

Alternative use of Buffered Channels

- An alternative syntax for message send/receive involves brackets:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

$\Longrightarrow$ used to highlight the first field, **e.g.** when it acts as 'message type'

- If - at the receiving side - some parameter is set to a constant value:

```
qname?const1,var2,var3
```

then the process blocks if the channel is empty or the input message field does not match the fixed constant value.

$\Longrightarrow$ used to filter messages

eval

It is **also** possible to filter incoming messages based on the value of a **variable** using the **eval** function. **e.g.:**

```
qname?eval(var1),var2,var3
```

Synchronous Channels

- A synchronous channel (aka rendezvous port) has size zero.

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

Synchronous Channels

- A synchronous channel (aka rendezvous port) has size zero.

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

- Messages can be exchanged, but not stored!

Synchronous Channels

- A synchronous channel (aka rendezvous port) has size zero.

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

- Messages can be exchanged, but not stored!
- Synchronous execution: a process executes a send at the same time another process executes a receive (as a single atomic operation).

Example:

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

active proctype A() {
    byte x = 124;
    printf("Send %d\n", x);
    name!msgtype(x);
    x = 121
    printf("Send %d\n", x);
    name!msgtype(x);
}
```

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

Channels of channels

- Message parameters are always passed **by value**.
- We can also pass the value of a channel from a process to another.

Example:

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

active proctype A() {
        chan loc = [0] of { mtype, byte };
        glob!loc;           /* send channel loc through glob */
        loc?msgtype(121)    /* read 121 from channel loc     */
}
active proctype B() {
        chan who;
        glob?who;           /* receive channel loc from glob */
        who!msgtype(121)    /* write 121 on channel loc      */
}
```

Q: what if B sends 122 on channel loc?

Channels of channels

- Message parameters are always passed **by value**.
- We can also pass the value of a channel from a process to another.

Example:

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

active proctype A() {
        chan loc = [0] of { mtype, byte };
        glob!loc;           /* send channel loc through glob */
        loc?msgtype(121)    /* read 121 from channel loc     */
}
active proctype B() {
        chan who;
        glob?who;           /* receive channel loc from glob */
        who!msgtype(121)    /* write 121 on channel loc      */
}
```

Q: what if B sends 122 on channel loc? both A and B are forever blocked

Channels and Ambiguity [1/2]

Example:

```
mtype = { MESSAGE };
chan in = [1] of { mtype };

active proctype A() {
    mtype m;
    if
        :: in?m ->
           printf("Message Received.\n");
        :: else ->
           printf("No Message.\n");
    fi
}

init {
    if
        :: true -> in!MESSAGE;
        :: true -> skip;
    fi
}
```

Q: how long should `A` wait before the `else` branch is taken?

Solution:

- use **message poll** to inspect the content of the channel

Example:

```
mtype = { MESSAGE };
chan in = [1] of { mtype };

active proctype A() {
    mtype m;
    if
        :: atomic { in?[m] -> in?m } ->
           printf("Message Received.\n");
        :: else ->
           printf("No Message.\n");
    fi
}

init {
    if
        :: true -> in!MESSAGE;
        :: true -> skip;
    fi
}
```

Sorted Send and Random Receive [1/2]

sorted send

- message is inserted immediately **before** the **oldest** message that succeeds it in numerical order
- syntax: `chname!!value`
- e.g.
  - `c!3; c!1; --> c([3, 1])`
  - `c!!3; c!!1; --> c([1, 3])`

random receive

- executable if there **exists** at least one message buffered in the message channel that can be received, **regardless of its position**
- syntax: `chname??value`
- e.g. given `c([3, 1])`
  - `c?1 --> blocks, 1 is not oldest element in queue`
  - `c??1 --> ok!`

Sorted Send and Random Receive [2/2]

```
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 is the sequence of printed values, for the following combinations?

- S1 + R1:
- S1 + R2:
- S2 + R1:
- S2 + R2:

Sorted Send and Random Receive [2/2]

```
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 is the sequence of printed values, for the following combinations?

- S1 + R1: (1,2) (1,1) (1,3) (0,1)
- S1 + R2:
- S2 + R1:
- S2 + R2:

Sorted Send and Random Receive [2/2]

```
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 is the sequence of printed values, for the following combinations?

- S1 + R1: (1,2) (1,1) (1,3) (0,1)
- S1 + R2: (1,1) (0,1)
- S2 + R1:
- S2 + R2:

Sorted Send and Random Receive [2/2]

```
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 is the sequence of printed values, for the following combinations?

- S1 + R1: (1,2) (1,1) (1,3) (0,1)
- S1 + R2: (1,1) (0,1)
- S2 + R1: (0,1) (1,1) (1,2) (1,3)
- S2 + R2:

Sorted Send and Random Receive [2/2]

```
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 is the sequence of printed values, for the following combinations?

- S1 + R1: (1,2) (1,1) (1,3) (0,1)
- S1 + R2: (1,1) (0,1)
- S2 + R1: (0,1) (1,1) (1,2) (1,3)
- S2 + R2: (0,1) (1,1)

Contents

Labels

end-state labels

- used to mark **valid end-states**, and tell them apart from a deadlock situations
- by **default**, the only valid end-state is reached when the process reaches the *syntactic end* of its body
- includes any label starting with `'end'`

progress-state labels

- used to mark a state that **must** be executed for the protocol/process to make progress
- any **infinite cycle** that does not cross a **progress state** is a potential starvation loop
- includes any label starting with `'progress'`

Contents

Basic verification

```
chan com = [0] of { byte };
byte value;
proctype p() {
    byte i;
    do
        :: if
              :: i >= 5 -> break
              :: else -> printf("Doing something else\n"); i ++
           fi
        :: com ? value; printf("p received: %d\n",value)
    od;
    ... /* fill in for formal verification */
}
init {
    run p();
    end: com ! 100;
}
```

Q: is it possible that process **p** does not read from the channel at all?

Basic verification

```
chan com = [0] of { byte };
byte value;
proctype p() {
    byte i;
    do
        :: if
               :: i >= 5 -> break
               :: else -> printf("Doing something else\n"); i ++
           fi
        :: com ? value; printf("p received: %d\n",value)
    od;
    ... /* fill in for formal verification */
}
init {
    run p();
    end: com ! 100;
}
```

Q: is it possible that process **p** does not read from the channel at all? Yes

Exercises [1/2]

- **Ex. 1:** write a PROMELA model that sums up an array of integers.
  - declare and (non-deterministically) initialize an integer array with values in [0, 9].
  - add a loop that sums even elements and subtracts odd elements.
  - visually check that it is correct.
  - **Q:** is it possible to initialize the array with a randomly chosen value among any valid integer? how?
- **Ex. 2:** declare a synchronous channel and create two processes:
  - The first process sends the characters 'a' through 'z' onto the channel.
  - The second process reads the values of the channel and outputs them as characters.
  - Check if sooner or later the second process will read the letter 'z'.
- **Ex. 3:** replace the synchronous channel with a buffered channel and check how the behaviour changes.

Exercises [2/2]

- **Ex. 4:** explain why *Produced 0* can appear twice in a row simulating:

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

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

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

Hints:

- add a global variable `last` initialized to -1
- assert `last != _pid` after each `printf` statement
- assign `_pid` to `last` just before releasing the turn
- use spin to look for a trace that falsifies the assertion

  $\Longrightarrow$ use `spin -search -bfs buggy.pml`
- replay the counter-example

  $\Longrightarrow$ use `spin -t -p -l -g`

Q: how would you fix the code?