An Overview of PROMELA*

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

Formal Methods Lab Class, Mar 03, 2015

Università degli Studi di Trento

---

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani for FM lab 2005/14

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

Contents

Process Initialization [1/3]

- $init$ is always the first process initialized (if declared);

Process Initialization [1/3]

- *init* is always the first process initialized (if declared);
- **active**: process created at initialization phase (right after **init**)

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

Process Initialization [1/3]

- *init* is always the first process initialized (if declared);
- **active**: process created at initialization phase (right after **init**)

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

- **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`

If present, active process parameters 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);
}
```

If present, active process parameters default to 0.

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

- A process does not necessarily start right after being created

```
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 (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 (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).
- Statements are atomic: each statement is executed without interleaving with other processes.
- 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 counter **_pid** (location within the proctype);
  - value of the local variables.
- A process communicates with other processes:
  - using global (shared) variables (might need synchronization!);
  - using channels.

Statements [1/2]

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

Statements [1/2]

- Every statement is either *executable* or *blocked*.
- Always executable:
  - print statements
  - assignments
  - skip
  - assert
  - break
  - ...

Statements [1/2]

- 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;
  - expressions

Statements [2/2]

- 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).

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.
- An array variable can be assigned with an array of values (e.g. `a = {1, 2, 3}`) only within a process body (it does not work at global scope).

Data structures

- A **run** statement accepts a list of variables or structures, but no array.

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

Note: but array can still be enclosed in data structures

- 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`:

```
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`:

```
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`:

```
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)
  ```

  It can be used to highlight that the first message field is interpreted 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)
  ```

  It can be used to highlight that the first message field is interpreted 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.

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).

```
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.

```
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.

```
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

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

- **Ex. 1:** write a PROMELA model that sums up an array of integers.
  - declare and (non-deterministically) initialize an integer array.
  - add a loop that sums up the elements.
  - visually check that it is correct.

- **Ex. 2:** declare a synchronous channel and create two processes:
  - The first process sends the numbers 0 through 9 onto the channel.
  - The second process reads the values of the channel and outputs them.
  - Check if sooner or later the second process will read the number 9.

- **Ex. 3:** replace the synchronous channel with a buffered channel and check how the behaviour changes.