An Overview of PROMELA[*]

Alessandra Giordani

agiordani@disi.unitn.it

http://disi.unitn.it/~agiordani

Formal Methods Lab Class, October 4, 2012

UNIVERSITÀ DEGLI STUDI DI TRENTO

[*] These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le for FM lab 2011/13

Attention

- New timetable: Lab Class on Fridays 9.20-10.50
- There will be no lab lesson next week (on March 14th)
- When do you want to recover the class?
  - March 26th afternoon (14.00-16.00)
  - April 1st, 2nd or 3rd?

Contents

Promela

- Promela design is focused on process interaction at the system level
- Consequent features:
  - non-deterministic control structures,
  - primitives for process creation,
  - primitives for interprocess communication.
- Consequent lacks:
  - functions with return values,
  - expressions with side-effects,
  - data and functions pointers.

PROMELA is a language for building verification models.
(not a programming language!)

Types of objects

Three basic types of objects:

- processes
- data objects
- message channels

Contents

Process Initialization

- By means of **active** (instantiate an initial set of processes):

Process Initialization

- By means of **active** (instantiate an initial set of processes):

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

Process Initialization

- By means of **active** (instantiate an initial set of processes):

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

- By means of **run** (creating new processes):

Process Initialization

- By means of **active** (instantiate an initial set of processes):

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

- By means of **run** (creating new processes):

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

Notes

- We cannot pass parameter values to $init$ or to active processes.

Notes

- We cannot pass parameter values to $init$ or to active processes.
- A newly created process may not start right after its initialization.

Notes

- We cannot pass parameter values to $init$ or to active processes.
- A newly created process may not start right after its initialization.
- To keep the system finite, only 255 processes can be alive in the same moment.

Notes

- We cannot pass parameter values to $init$ or to active processes.
- A newly created process may not start right after its initialization.
- To keep the system finite, only 255 processes can be alive in the same moment.
- A process “terminates” when it reaches the end of its code.
- A process “dies” when it has terminated and all processes instantiated later have died.

Process Execution

- A process executes 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 rendezvous communication).
- 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).

Contents

Variable Scope

- There are only two levels of scope:
  - global: if it is declared outside all process declarations,
  - process local: if it is declared within a process declaration.

Variable Scope

- There are only two levels of scope:
  - global: if it is declared outside all process declarations,
  - process local: if it is declared within a process declaration.
- Spin Version 6 (or newer) limits the scope of a variable to the block in which it is declared.

```
init {  /* x declared in outer block */
        int x;
        {       /* y declared in inner 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);
}
```

Variable declarations are implicitly moved to the beginning of the process.

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 .. $2^{n}-1$

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 .. $2^{n}-1$

- No character type: literal character values can be assigned to variables of type `byte` and printed using the %c format specifier.
- No string variables: messages can be modeled using numeric codes.
- No floating-point data types: exact values are not important!

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  */
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 indicing starts at 0.

Data structures

```
typedef Field {
        short f = 3;
        byte  g
};
typedef Record {
        byte a[3];
        int fld1;
        Field fld2;
        chan p[3];
        bit b
};
proctype me(Field z) { z.g = 12 }
init { Record goo; Field  foo;
        run me(foo)
}
```

Arrays and Data structures

- A structure can be passed as argument to a **run** statement, provided it contains no arrays. (In the example, foo can be passed, goo cannot.)
- Multi-dimensional arrays are not supported, although there are indirect ways:

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

Array a[4];
```