T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 3

25th of September 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Arrays and Records

Array indices in Promela start at 0. No multidimensional arrays. Records (C style structs) are available through the `typedef` keyword:

```
typedef foo {
  short f1;
  byte f2;
}

foo rr; /* variable declaration */
rr.f1 = 0;
rr.f2 = 200;
```


Variables and Types

- Variables need to be declared
- Variables can be given value by:
  - Assignment
  - Argument passing (input parameters to processes)
  - Message passing
- Variables have exactly two scopes: global and process local variables

Data Manipulation

Most of C language arithmetic, relational, and logical operations on variables are supported in Spin with the same syntax (including comparison operators, bitshifts, masking etc.)

When in doubt, try the “C” way of doing things and you will probably be right.

Data Manipulation, Example

```
active[1] proctype foo() {
  int c,d;
  printf("c:%d d:%d\n", c, d);
  c++;
  c++;
  d = c+1;
  d = d<<1;
  c = c*d;
  printf("c:%d d:%d\n", c, d);
  c = c&3;
  d = d/5;
  printf("c:%d d:%d\n", c, d);
}
```


Data Manipulation, Example

Running the example we get:

```
$ spin ex4.pml
      c:0 d:0
      c:12 d:6
      c:0 d:1
1 process created
```


Conditional Expressions

C-style conditional expressions have to be replaced:

```
active[1] proctype foo() {
  int a,b,c,d;
  b=1;c=2;d=3;
#if 0
  a = b ? c : d;         /* not valid in Promela! */
  a = b -> c : d;        /* not valid in Promela! */
#endif
  a = (b -> c : d);      /* valid in Promela */
  printf("a:%d\n", a);
}
```


Conditional Expressions (cnt.)

The parenthesis in "`(foo -> bar : baz)`" are vital! The expression "`foo -> bar : baz`" will generate a syntax error!

```
$ spin ex5.pml
      a:2
1 process created
```

Promela Statements

- The body of a process consists of a sequence of statements
- A statement can in current global state of the model either be:
  - Executable: the statement can be executed in the current global state
  - Blocked: the statement cannot be executed in the current global state
- Assignments are always executable
- An expression is executable if it evaluates to non-zero (true)

Executable Statements

```
0<1;     /* Always executable */
x<5;     /* Executable only when x is smaller than 5 */
3+x;     /* Executable if x is not -3 */
(x > 0 && y > x); /* Executable if x > 0 and y > x */
                  /* Note: This is a single, atomic
                     statement!                    */
```


Statements

- The `skip` statement is always executable. It does nothing but changes the value of the program counter
- The `run` statement is executable if a new process can be created (recall the 255 process limit)
- The `printf` statement is always executable (it is used only for simulations, not in model checking)

Statements (cnt.)

```
assert(<expr>);
```

- The `assert` statement is always executable
- If `<expr>` evaluates to zero, Spin will exit with an error
- The assert statements are handy for checking whether certain properties hold in the current global state of the model

Intuition of the Promela Semantics

- Promela processes execute in parallel
- Non-deterministic scheduling of the processes
- Processes are interleaved - statements of concurrently running processes cannot occur simultaneously
- All statements are atomic - each statement is executed without interleaving of other processes
- Each process can be non-deterministic - have several executable statements enabled. Only one statement is selected for execution nondeterministically

The `if`-statement

Now we proceed to non-atomic compound statements.
The `if` statement is also called the selection statement and has gotten its syntax from Dijkstra's guarded command language.
Example:

```
chan STDIN;
active[1] proctype foo() {
  int c;
  STDIN?c; /* Read a char from standard input */
  if
    :: (c == -1) -> skip; /* EOF */
    :: ((c % 2) == 0) -> printf("Even\n");
    :: ((c % 2) == 1) -> printf("Odd\n");
  fi
}
```


Example: `if`-statement

```
$ spin ex6.pml
a
        Odd
1 process created
$ spin ex6.pml
b
        Even
1 process created
$ spin ex6.pml
1 process created
```


The `if`-statement (cnt.)

The `if`-statement has the general form:

```
if
  :: (choice_1) -> statement_1_1; statement_1_2; ...
  :: (choice_2) -> statement_2_1; statement_2_2; ...
  :: ...
  :: (choice_n) -> statement_n_1; statement_n_2; ...
fi
```

The `if`-statement (cnt.)

- The `if`-statement is executable if there is a `choice_i` statement which is executable. Otherwise i is blocked.
- If several `choice_i` statements are executable, Spin non-deterministically chooses one to be executed.
- If `choice_i` is executed, the execution then proceeds to executing `statement_i_1; statement_i_2; ... statement_i_m;`
- After this the program continues from the next statement after the `fi`

Example 2: `if`-statement

An `else` branch is taken only if none of `choice_i` is executable

```
active[10] proctype foo() {
  pid p = _pid;
  if
    :: (p > 2) -> p++;
    :: (p > 3) -> p--;
    :: else -> p = 0;
  fi;
  printf("Pid:%d, p:%d\n", _pid, p)
}
```


Example 2: `if`-statement (cnt.)

```
$ spin -T ex7.pml
Pid:7, p:8
Pid:0, p:0
Pid:3, p:4
Pid:9, p:8
Pid:6, p:7
Pid:4, p:3
Pid:1, p:0
Pid:5, p:6
Pid:2, p:0
Pid:8, p:9
10 processes created
```


The `do`-statement

The way of doing loops in Promela

- With respect to choices, a `do` statement behaves same way as an `if`-statement
- However, after one selection has been made the `do`-statement repeats the choice selection
- The (always executable) `break` statement can be used to exit the loop and continue from the next statement after the `od`

The do-statement (cnt.)

The do-statement has the general form:

```
do
  :: (choice_1) -> statement1_1; statement1_2; ...
  :: (choice_2) -> statement2_1; statement2_2; ...
  :: ...
  :: (choice_n) -> statementn_1; statementn_2; ...
od
```


Example: For loop

```
active[1] proctype foo() {
  int i = 0;
  do
    :: (i < 10) -> printf("i: %d\n",i); i++;
    :: else -> break
  od
}
```


Example: For loop (cnt.)

```
$ spin ex8.pml
      i: 0
      i: 1
      i: 2
      i: 3
      i: 4
      i: 5
      i: 6
      i: 7
      i: 8
      i: 9
1 process created
```


Example: Euclid

```
proctype Euclid(int x, y)
{
  do
  :: (x >  y) -> x = x - y
  :: (x <  y) -> y = y - x
  :: (x == y) -> break
  od;
  printf("answer: %d\n", x)
}

init { run Euclid(38, 14) }
```

Example: Euclid (cnt.)

Running the algorithm we get:

```
$ spin euclid.pml
          answer: 2
2 processes created
```


Example: Infamous goto-statement

```
proctype Euclid(int x, y)
{
  do
  :: (x >  y) -> x = x - y
  :: (x <  y) -> y = y - x
  :: (x == y) -> goto done
  od;
done:
  printf("answer: %d\n", x)
}
init { run Euclid(38, 14) }
```


Communication

- Message passing through message channels (first-in first-out (FIFO) queues)
- Rendezvous synchronization (handshake). Syntactically appears as communication over a channel with capacity zero

Both are defined by channels:

```
chan <chan_name> = [<capacity>] of
{<t_1>, <t_1>,..., <t_n>};
```

where t_i are the types of the elements transmitted over the channel.

Sending Messages

Consider the case where ch is a channel with capacity ≥ 1

- The send-statement:

  ```
  ch ! <expr_1>, <expr_2>,..., <expr_n>;
  ```

  - Is executable only if the channel is not full
  - Puts a message at the end of the message channel ch
  - The message consists of a tuple of the values of the expressions <expr_i> - the types should match the channel declaration

Receiving Messages

Consider the case where ch is a channel with capacity ≥ 1

- The receive-statement:

```
ch ? <var_1>, <var_2>,..., <var_n>;
```

  - Is executable only if the channel is not empty
  - Receives the first message of the message channel ch and fetches the individual fields of the vars into variables <var_i> - the types should match the channel declaration
  - Any of the <var_i> can be replaced by a constant. In that case the statement is executable only if the first message matches the constants.

Example: Alternating Bit Protocol

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };
active proctype Sender()
{
again:
  to_rcvr!msg1;
  to_sndr?ack1;
  to_rcvr!msg0;
  to_sndr?ack0;
  goto again
}
```


Example: Alternating Bit Protocol

```
active proctype Receiver()
{
again:
  to_rcvr?msg1;
  to_sndr!ack1;
  to_rcvr?msg0;
  to_sndr!ack0;
  goto again
}
```


Example: Alternating Bit Protocol

```
$ spin -c -u10 alternatingbit.pml
proc 0 = Sender
proc 1 = Receiver
q\p   0   1
  1   to_rcvr!msg1
  1   .   to_rcvr?msg1
  2   .   to_sndr!ack1
  2   to_sndr?ack1
  1   to_rcvr!msg0
  1   .   to_rcvr?msg0
  2   .   to_sndr!ack0
  2   to_sndr?ack0
-------------
depth-limit (-u10 steps) reached
-------------
```

Advanced Promela

- Promela is somewhat like the C language - very powerful but at the same time hard to fully master
- In the following we discuss more advanced modelling features of Promela

Alternative Send/Receive Syntax

- Alternative syntax for the send-statement:

```
ch ! <expr_1> (<expr_2>,..., <expr_n>);
```

- Alternative syntax for the receive-statement:

```
ch ? <var_1> (<var_2>,..., <var_n>);
```


More Promela Message Passing

- Peeking at the message channel can be implemented in Promela with:

```
ch ? [<var_1>, <var_2>,..., <var_n>];
```

  It is executable iff the message received would be but does not actually remove the message from the channel. Moreover, the contents of the variables `<var_i>` remain unchanged.
- To do the same except that this time the variables `<var_i>` are changed, use:

```
ch ? <<var_1>, <var_2>,..., <var_n>>;
```

  For example, `ch ? <x,y>` puts the contents of the first message in the channel `ch` to vars `x` and `y` without removing the message from the channel.

Other Channel Operations

- `len(ch)` - returns the number of messages in channel `ch`
- `empty(ch)` - returns true if `ch` is empty, otherwise returns false
- `nempty(ch)` - returns true if `ch` is not empty, otherwise returns false
- `full(ch)` - returns true if `ch` is full, otherwise returns false
- `nfull(ch)` - returns true if `ch` is not full, otherwise returns false

Rendezvous Communication

- In Promela the synchronization between two processes (rendezvous) is syntactically implemented as message passing over a channel of capacity 0.
- In this case the channel cannot store messages, only pass immediately from the sender to the receiver.

Rendezvous Example

```
mtype = { msgtype };

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

active proctype A()
{       name!msgtype(124);    /* Alternative syntax */
        name!msgtype(121)     /* used here */
}

active proctype B()
{       byte state;
        name?msgtype(state)   /* And here */
}
```


Rendezvous Example (cnt.)

- The processes A and B in the example synchronize: The execution of both the send and the receive is blocked until a matching send/receive pair becomes enabled.
- When a matching send/receive pair is enabled, they can execute and communicate in an atomic step the sent message from the sender to the receiver.
- Note that if the channel had a capacity of 2 in the example, the process A could already terminate before the process B starts executing.

Executability of Statements (recap)

- `skip` - always executable
- `assert(<expr>)` - always executable
- `<expression>` - executable if not zero
- `<assignment>` - always executable
- `if` - executable if at least one guard is
- `do` - executable if at least one guard is
- `break` - always executable
- send `ch ! msg` - executable if channel `ch` is not full
- receive `ch ? var` - executable if channel `ch` is not empty

Advanced Promela - `atomic`

In Promela a sequence of statements can be grouped together to execute atomically by using the `atomic` compound statement:

```
atomic { /* Swap values of a and b */
   tmp = b;
   b = a;
   a = tmp
}
```


Atomic Sequences (cnt.)

- The sequence of statements inside an `atomic` sequence execute together in an uninterrupted manner
- In other words no other process can be scheduled until the atomic sequence has been completed
- In the example that means that no other process can be run to see the state where both `a` and `b` contains the old value of `a`

Atomic - Examples

- The following Promela statement sequences are not atomic:

```
nfull(ch) -> ch!msg0;   /* Not atomic! */
ch?[msg0] -> ch?msg0;   /* Not atomic! */
```

- They can be replaced by:

```
atomic { nfull(ch) -> ch!msg0 };/* Atomic! */
ch?msg0;                      /* Trivially atomic! */
```


Atomic Sequences - Details

- The atomic sequences are also allowed to contain branching and non-determinism
- If any statement inside an atomic sequence is found to be unexecutable (i.e., it blocks the execution), other processes are allowed to run
- The states reached inside an `atomic` sequence still exists in the statespace of the system, not only the last state reached by the execution