T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 3

2006.02.10

Keijo Heljanko

Keijo.Heljanko@tkk.fi

The Spin Model Checker

- The model checker Spin was designed at Bell Labs by Gerard J. Holzmann (currently at NASA)
- It received the ACM Software System award in 2002. (Other winners: Unix, TeX, Smalltalk, Postscript, TCP/IP)
- Originally designed for data-communications protocol analysis
- The modelling language of Spin is called Promela
- The Spin Website has more material: `http://www.spinroot.com/`

Spin

Some of the reasons why Spin is successful

- Very efficient implementation (using generated C code)
- Contains advanced model checking algorithms, several of which are enabled by default
- A graphical user interface available (Xspin)
- Has been around for a while (15 years) and has been solidly supported by Holzmann

The Acronyms

- Spin = (Simple Promela Interpreter)
- Promela = (Protocol/Process Meta Language)

The Books

- The version 1.0 of Spin was published in Jan 1991:
  - Gerard J. Holzmann: Design and Validation of Computer Protocols, Prentice Hall, Nov 1990.
  - Book still available as PDF from:
    `http://spinroot.com/spin/Doc/Book91.html`
- A new book on Spin is much more up to date (v. 4.x):
  - Gerard J. Holzmann: The Spin Model Checker - Primer and Reference Manual, Addison-Wesley, Sep 2003, ISBN 0-321-22862-6.
  - For Book extras see:

    `http://spinroot.com/spin/Doc/Book_extras/index.html`

Promela

- The input language of the Spin model checker
- Control flow syntax inherited from Dijkstra's guarded command language
- Message passing primitives from Hoare's CSP language
- Syntax for data manipulation from Kernighan and Ritchie's C language

Modelling in Promela

This part is based on a nice Spin Beginners' Tutorial by Theo C. Ruys:
`http://spinroot.com/spin/Doc/SpinTutorial.pdf`
and The Spin Model Checker - Primer and Reference Manual

A Promela model consists of a set of processes communicating with each other through:

- Global variables
- Message queues of fixed capacity (called channels in Promela)
- Synchronization (rendezvous) on common actions

Promela Model

A Promela model consists of:

- Type declarations
- Channel declarations
- Variable declarations
- Process declarations
- Optionally: the `init` process (the “main()” process)

State of a Promela Model

The state of a Promela model consists of states of:

- Running processes (program counter)
- Data objects (global and local variables)
- Message channels

Finite State Models

Promela models are always finite state because

- All data objects have a bounded domain
- All message channels have a bounded capacity
- The number of running processes is limited (max 255 processes)
- The number of Promela statements in each process is finite - Also no procedure mechanism exists

Thus analysis of Promela models is in theory decidable.

In practice the available memory and time is the limit.

Processes

A process type (`proctype`) consists of

- Name - name of the proctype
- List of formal parameters - inputs given at start
- Local variable declarations
- Body - a sequence of statements: code of the procedure

Processes (Example)

In the following code the init process runs two instances of the you_run proctype

```
proctype you_run(byte x)
{
        printf("x = %d, pid = %d\n", x, _pid)
}

        /* leaving pids implicit */
init {
        run you_run(0);
        run you_run(1)
}
```

Processes (Example cnt.)

We can use spin for random simulation as follows:

```
$ spin ex1.pml
          x = 0, pid = 1
              x = 1, pid = 2
3 processes created
$ spin ex1.pml
                x = 1, pid = 2
          x = 0, pid = 1
3 processes created
$ spin ex1.pml
          x = 0, pid = 1
          x = 1, pid = 1
3 processes created
```

Processes (Example cnt.)

Note that Spin used indentation to show which process printed what. (You can use `spin -T` to disable this.) You can provide a seed to the Spin pseudorandom number generator as follows:

```
$ spin -n5 ex1.pml
          x = 0, pid = 1
              x = 1, pid = 2
3 processes created
```

In Promela the `init` process gets alway the pid `0` but the other processes dynamically allocate their pids

Process

- Is defined by `proctype` definition
- executes concurrently with all other processes, the scheduling used is completely non-deterministic
- There may be several processes of the same type
- Local state:
  - Program counter
  - Contents of local variables

Creating Processes

- Processes are created using the `run` statement.
  To be precise: `run` expression (with a side-effect).
- Processes can also be created at the startup by adding `active[numprocs]` in front of a `proctype Foo()` to create `numprocs` instances of `proctype Foo`

Example:

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

Creating Processes (cnt.)

Running the example:

```
$ spin ex2.pml
            my pid is: 1
        my pid is: 0
2 processes created

$ spin ex2.pml
        my pid is: 0
            my pid is: 1
2 processes created
```

Variables and Types

The Promela basic types are (sizes match those of C).

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`	$0..2^{32}-1$

Example Declarations

```
bit x, y;                    /* two single bits, initially 0 */
bool turn = true;          /* boolean value, initially true */
byte a[12];          /* array of 12 bytes initialised to 0 */
short b[4] = 89;  /* array of 4, all initialised to 89 */
int cnt = 67;                  /* integer initialised to 67 */
unsigned v : 5;                /* unsigned stored in 5 bits */
unsigned w : 3 = 5;    /* value range 0..7, initially 5 */
mtype n; /* uninitialised mtype (enumeration) variable */

chan in = [3] of {short, byte, bool}; /* message channel
with 3 messages capacity, messages have three fields   */
```

Mtype

The `mtype` (message type) keyword is a way of introducing enumerations in Spin.

Example:

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
        mtype n = pear; /* initialise n to pear */

        printf("the value of n is %e\n", n)
}
```

Mtype (cnt.)

Running the example in Spin:

```
$ spin ex3.pml
      the value of n is pear
1 process created
```

Arrays and Records

Array indices 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 */
  a = b -> c : d;         /* not valid */
#endif
  a = (b -> c : d);       /* valid */
  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_3) -> statement_3_1; statement_3_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_n;`
- 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_3) -> statement3_1; statement3_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> = [<capasity>] 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 capasity ≥ 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 capasity ≥ 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
  - An 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
-------------
```