The Spin Model Checker : Part I

Moonzoo Kim
KAIST
Hierarchy of SW Coverage Criteria

(SW) Model checking

Concolic testing
active type A() {
    byte x;
    again:
        x++;
        goto again;
}

active type A() {
    byte x;
    again:
        x++;
        goto again;
}

active type B() {
    byte y;
    again:
        y++;
        goto again;
}
A few characteristics of Spin

- Promela allows a finite state model only
- Asynchronous execution
- Interleaving semantics for concurrency
- 2-way process communication
- Non-determinism
- Promela provides (comparatively) rich set of constructs such as variables and message passing, dynamic creation of processes, etc.
/*
 * Model of cell-phone handoff strategy in a mobile network.
 * A translation from the pi-calculus description of this
 * model presented in:
 * Fredrik Orava and Joachim Parrow, 'An algebraic verification
 * of a mobile network,' Formal aspects of computing, 4:497-543 (199
 * 2).
 * For more information on this model, email: joachim@it.kth.se
 * See also the simplified version of this model in mobile2
 * to perform the verification:
 * $ spin -a mobile1
 * $ cc -o pan pan.c
 * $ pan -a
 */

mtype = { data, ho_cmd, ho_com, ho_acc, ho_fail, ch_rel, white, red, blue };
byte a_id, p_id; /* ids of processes refered to in the property */

3 gcc -o pan pan.c
4 ./pan -D | dot > dot.tmp
5 C:/spin/mobile1.pml:1
6 spin -o3 -a mobile1.pml
ltiltl_0: ( !( !(! ( <> (((BS[a_id]@progress)) || ((BS[p_id]@progress))))))) || ( [] ( !( <> (inp?[red]))) || ( <> (out?[red]))) )
7 gcc -o pan pan.c
8 ./pan -D | dot > dot.tmp
Tcl GUI of SPIN (ispin.tcl): Verification Window

```
byte x=0, y=0, z =0;
active proctype p() {
  x=y+1;
  y=z+1;
  z = x+1;
}
active proctype q() {
  y=z+1;
  z=x+1;
  x=y+1;
}

int {
  timeout->printf("x=%d,y=%d,z=%d",x,y,z);
  assert(0);
}
/*
spin -a 2threads.pml
```
Tcl GUI of SPIN (ispin.tcl): Simulation Window

Spin Version 6.2.7 -- 2 March 2014 :: iSpin Version 1.1.0 -- 7 June 2012

2threads.pml

Mode

- Random, with seed: 123
- Interactive (for resolution of all nondeterminism)
- Guided, with trail: C:/Dropbox/classes/Spring14-cs492B/2threads/2threads.pml5.trail

A Full Channel

- blocks new messages
- loses new messages
- MSC + stmtt

Output Filtering (reg. exps.)

- process ids:
- queue ids:
- var names:
- tracked variable:
- track scaling:

Initial steps skipped: 0
Maximum number of steps: 10000
MSC max text width: 20
MSC update delay: 25

Save in: msc.ps

Variable values, step 1

| y = 1 |

Queues, step 1

| q:1:1 |

proc 1 (q:1) 2threads.pml:10 (state 1) [y = (z+1)]
proc 0 (p:1) 2threads.pml:4 (state 1) [x = (y+1)]
proc 1 (q:1) 2threads.pml:11 (state 2) [z = (x+1)]
proc 0 (p:1) 2threads.pml:5 (state 2) [y = (z+1)]
proc 1 (q:1) 2threads.pml:12 (state 3) [x = (y+1)]
proc 0 (p:1) 2threads.pml:6 (state 3) [z = (x+1)]
proc 2 (init:1) 2threads.pml:16 (state 1) (timeout)
proc 2 (init:1) 2threads.pml:18 (state 2) (timeout)
proc 2 (init:1) 2threads.pml:17 (state 3) (timeout)

spin: 2threads.pml:17, Error: assertion violated
spin: text of failed assertion: assert(0)

#processes: 3
Overview of the Promela

byte x;
chan ch1 = [3] of {byte};

active[2] proctype A() {
    byte z;
    printf("x=%d\n", x);
    z = x + 1;
    ch1!z
}

proctype B(byte y) {
    byte z;
    ch1?z;
}

Init {
    run B(2);
}

Global variables (including channels)

Process (thread) definition and creation

Similar to C syntax but simplified
- No pointer
- No real datatype such as float or real
- No functions

Processes are communicating with each other using
- Global variables
- Message channels

Process can be dynamically created

Scheduler executes one process at a time using interleaving semantics
active[2] proctype A() {
    byte x;
    printf("A%d is starting\n");
}

proctype B() {
    printf("B is starting\n");
}

Init {
    run B();
}

run() operator creates a process and returns a newly created process ID

There are 6 possible outcomes due to non-deterministic scheduling
- A0.A1.B, A0.B.A1
- B.A0.A1, B.A1.A0

In other words, process creation may not immediately start process execution
Variables and Types

- Basic types
  - bit
  - bool
  - Byte (8 bit unsigned integer)
  - short (16 bits signed integer)
  - Int (32 bits signed integer)

- Arrays
  - bool x[10];

- Records
  - typedef R { bit x; byte y;}

- Default initial value of variables is 0

- Most arithmetic (e.g., +,-), relational (e.g. >,==) and logical operators of C are supported
  - bitshift operators are supported too.
Promela spec generates only a finite state model because:

- Max # of active process $\leq 255$
- Each process has only finite length of codes
- Each variable is of finite datatype
- All message channels have bounded capability $\leq 255$
Each Promela statement is either executable:

- Blocked

There are six types of statement:

- Assignment: always executable
  - Ex. \( x = 3 + x, x = \text{run } A() \)
- Print: always executable
  - Ex. \texttt{printf(“Process \%d is created.\n”, _pid);} 
- Assertion: always executable
  - Ex. \texttt{assert( x + y == z)}
- Expression: depends on its value
  - Ex. \( x + 3 > 0, 0, 1, 2 \)
  - Ex. \texttt{skip, true}
- Send: depends on buffer status
  - Ex. \texttt{ch1!m} is executable only if \texttt{ch1} is not full
- Receive: depends on buffer status
  - Ex. \texttt{ch1?m} is executable only if \texttt{ch1} is not empty
An expression is also a statement

- It is executable if it evaluates to non-zero
  - 1 : always executable
  - 1<2:always executable
  - x<0: executable only when x < 0
  - x-1: executable only when x !=0

If an expression statement in blocked, it remains blocked until other process changes the condition

- an expression e is equivalent to while(!e); in C
assert (expr)

- assert is always executable
- If expr is 0, SPIN detects this violation
- assert is most frequently used checking method, especially as a form of invariance

- ex. active proctype inv() { assert( x== 0);}
  - Note that inv() is equivalent to [] (x==0) in LTL with thanks to interleaving semantics
Generation of all possible interleaving scenarios

Original execution tree

After adding Inv() process

Therefore, just a single assert(x==0) statement in Inv() can check if x==0 all the time
Promela provides low-level control mechanism, i.e., goto and label as well as if and do.

Note that non-deterministic selection is supported.

else is predefined variable which becomes true if all guards are false; false otherwise.

```plaintext
proctype A() {
    byte x;
    starting:
    x= x+1;
    goto starting;
}
```

```plaintext
proctype A() {
    byte x;
    if
        ::x<=0 -> x=x+1
        ::x==0 -> x=1
    fi
}
```

```plaintext
proctype A() {
    byte x;
    do
        :: x<=0 ->x=x+1;
        :: x==0 ->x=1;
        :: else -> break
    od
}
```

```plaintext
int i;
for (i : 1 .. 10) {
    printf("i =%d\n",i)
}
```
bool lock;
byte cnt;

active[2] proctype P() {
    !lock -> lock=true;
    cnt=cnt+1;
    printf("%d is in the crt sec!\n",_pid);
    cnt=cnt-1;
    lock=false;
}

active proctype Invariant() {
    assert(cnt <= 1);
}

spin -a crit.pml
[...]
pan: assertion violated (cnt<=1) (at depth 8)
pan: wrote crit.pml.trail

Full statespace search for:
never claim - (none specified)
assertion violations +
acceptance cycles - (not selected)
invalid end states +

State-vector 36 byte, depth reached 16, errors: 1
119 states, stored
47 states, matched
166 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
4.879 memory usage (Mbyte)
[...]
Critical Section Example (cont.)

[root@moonzoo spin_test]# spin -t -p crit.pml
Starting P with pid 0
Starting P with pid 1
Starting Invariant with pid 2
1: proc 1 (P) line  5 "crit.pml" (state 1)  [(!(lock))]
2: proc 0 (P) line  5 "crit.pml" (state 1)  [(!(lock))]
3: proc 1 (P) line  5 "crit.pml" (state 2)  [lock = 1]
4: proc 1 (P) line  6 "crit.pml" (state 3)  [cnt = (cnt+1)]
    1 is in the crt sec!
5: proc 1 (P) line  7 "crit.pml" (state 4)  [printf('%d is in the crt sec!\n',_pid)]
6: proc 0 (P) line  5 "crit.pml" (state 2)  [lock = 1]
7: proc 0 (P) line  6 "crit.pml" (state 3)  [cnt = (cnt+1)]
    0 is in the crt sec!
8: proc 0 (P) line  7 "crit.pml" (state 4)  [printf('%d is in the crt sec!\n',_pid)]
spin: line  13 "crit.pml", Error: assertion violated
spin: text of failed assertion: assert((cnt<=1))
9: proc 2 (Invariant) line 13 "crit.pml" (state 1)  [assert((cnt<=1))]
spin: trail ends after 9 steps
#processes: 3
    lock = 1
    cnt = 2
9: proc 2 (Invariant) line 14 "crit.pml" (state 2) <valid end state>
9: proc 1 (P) line  8 "crit.pml" (state 5)
9: proc 0 (P) line  8 "crit.pml" (state 5)
3 processes created
bool lock;
byte cnt;

active[2] proctype P() {
    atomic{ !lock -> lock=true;}
cnt=cnt+1;
printf("%d is in the crt sec!\n",_pid);
cnt=cnt-1;
lock=false;
}

active proctype Invariant() {
    assert(cnt <= 1);
}

[root@moonzoo revised]# a.out
Full statespace search for:
    never claim             - (none specified)
    assertion violations    +
    acceptance cycles       - (not selected)
    invalid end states      +

State-vector 36 byte, depth reached 14, errors: 0
62 states, stored
17 states, matched
79 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

4.879 memory usage (Mbyte)
Deadlocked Critical Section Example

bool lock;
byte cnt;

active[2] proctype P() {
    atomic{ !lock -> lock==true;}
    cnt=cnt+1;
    printf("%d is in the crt sec!\n",_pid);
    cnt=cnt-1;
    lock=false;
}

active proctype Invariant() {
    assert(cnt <= 1);
}

[[root@moonzoo deadlocked]# a.out
pan: invalid end state (at depth 3)

(Spin Version 4.2.7 -- 23 June 2006)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never claim             - (none specified)
    assertion violations    +
    acceptance cycles      - (not selected)
    invalid end states      +

State-vector 36 byte, depth reached 4, errors: 1
    5 states, stored
    0 states, matched
    5 transitions (= stored+matched)
    2 atomic steps
hash conflicts: 0 (resolved)

4.879 memory usage (Mbyte)
Deadlocked Critical Section Example (cont.)

[root@moonzoo deadlocked]# spin -t -p deadlocked_crit.pml
Starting P with pid 0
Starting P with pid 1
Starting Invariant with pid 2
  1: proc  2 (Invariant) line 13 "deadlocked_crit.pml" (state 1)
[assert((cnt<=1))]
  2: proc 2 terminates
  3: proc  1 (P) line  5 "deadlocked_crit.pml" (state 1)  [(!((lock)))]
  4: proc  0 (P) line  5 "deadlocked_crit.pml" (state 1)  [(!((lock)))]

spin: trail ends after 4 steps

#processes: 2
  lock = 0
  cnt = 0
  4: proc  1 (P) line  5 "deadlocked_crit.pml" (state 2)
  4: proc  0 (P) line  5 "deadlocked_crit.pml" (state 2)

3 processes created
Spin provides communications through various types of message channels:
- Buffered or non-buffered (rendezvous comm.)
- Various message types
- Various message handling operators

**Syntax**

```plaintext
chan ch1 = [2] of {bit, byte};

- ch1!0,10;ch1!1,20
- ch1?b,bt;ch1?1,bt

chan ch2 = [0] of {bit, byte}
```

Diagram:

```
Sender  (1,20)  (0,10)  Receiver
```

**Legend:**
- `Sender` refers to the sender of the message.
- `Receiver` refers to the receiver of the message.
- `(1,20)` and `(0,10)` represent the channels or tokens used in the communication.
## Operations on Channels

### Basic channel inquiry
- `len(ch)`
- `empty(ch)`
- `full(ch)`
- `nempty(ch)`
- `nfull(ch)`

### Additional message passing operators
- `ch?[x,y]`: polling only
- `ch?<x,y>`: copy a message without removing it
- `ch!!x,y`: sorted sending (increasing order)
- `ch??5,y`: random receiving
- `ch?x(y) == ch?x,y` (for user’s understandability)

### Be careful to use these operators inside of expressions
- They have side-effects, which spin may not allow
Faulty Data Transfer Protocol
(pg 27, data switch model proposed at 1981 at Bell labs)

mtype={init,ack, dreq, data, shutup, quiet, dead}
chan M = [1] of {mtype};
chan W = [1] of {mtype};

active proctype Mproc()
{
    W!init; /* connection */
    M?ack; /* handshake */

    timeout -> /* wait */
    if /* two options: */
        :: W!shutup; /* start shutdown */
        :: W!dreq; /* or request data */
        do
            :: M?data -> W!data
            :: M?data -> W!shutup;
            break
        od
    fi;
    M?shutup;
    W!quiet;
    M?dead;
}

active proctype Wproc()
{
    W?init; /* wait for ini*/
    M!ack; /* acknowledge */

    do /* 3 options: */
        :: W?dreq-> /* data requested */
            M!data /* send data */
        :: W?data-> /* receive data */
            skip /* no response */
        :: W?shutup->
            M!shutup; /* start shutdown*/
            break
    od;
    W!quiet;
    M!dead;
}
The Sieve of Eratosthenes (c. 276-196 BC)
Prints all prime numbers up to MAX

#define MAX 25
mtype = { number, eof }; chan root = [0] of { mtype, int };

init
{   int n = 2;
    run sieve(root, n);
    do 
        :: (n < MAX) -> n++; root!number(n)
        :: (n >= MAX) -> root!eof(0); break
    od
}

proctype sieve(chan c; int prime)
{
    chan child = [0] of { mtype, int };
    bool haschild; int n;
    printf("MSC: %d is prime\n", prime);
     end: do
        :: c?number(n) ->
        if
            :: (n%prime) == 0 -> printf("MSC: %d = %d*%d\n", n, prime, n/prime)
            :: else ->
                if
                    :: !haschild -> /* new prime */
                        haschild = true;
                        run sieve(child, n);
                    :: else ->
                        child!number(n)
                    fi;
        fi
        :: c?eof(0) -> break
    od;
    if
        :: haschild -> child!eof(0)
        :: else
    fi
[moonzoo@verifier spin]$ spin sieve-of-eratosthenes.pml

<table>
<thead>
<tr>
<th>Number</th>
<th>MSC: $\text{Number}$ is prime</th>
<th>MSC: $\text{Number}$ = $2 \times \text{Other Number}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>MSC: 2 is prime</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MSC: 3 is prime</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MSC: 4 = 2*2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MSC: 5 is prime</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MSC: 6 = 2*3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MSC: 7 is prime</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MSC: 8 = 2*4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MSC: 9 = 3*3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MSC: 10 = 2*5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MSC: 11 is prime</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>MSC: 12 = 2*6</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>MSC: 13 is prime</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>MSC: 14 = 2*7</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>MSC: 15 = 3*5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>MSC: 16 = 2*8</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>MSC: 18 = 2*9</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>MSC: 20 = 2*10</td>
<td></td>
</tr>
</tbody>
</table>