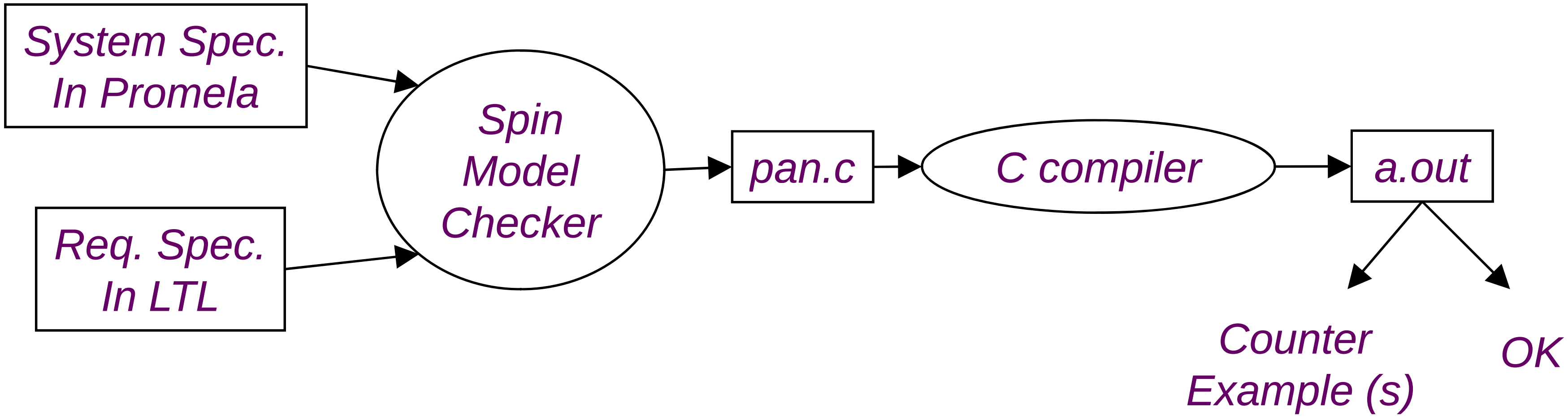
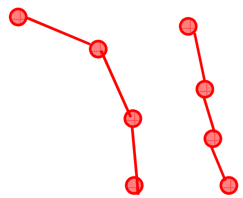
The Spin Model Checker : Part I

Moonzoo Kim

Provable SW Lab, KAIST

Overview of the Spin Architecture





- 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

Overview of the Promela

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

Global variables (including channels)

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

Process (thread) definition and creation

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

Another process definition

```
Init {
      run B(2);
}
```

System initialization

- 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

KAIST

Process Creation Example

```
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
  - A1.A0.B, A1.B.A0
  - 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 <= 255
  - Each process has only finite length of codes
  - Each variable is of finite datatype
  - All message channels have bounded capability <= 255

- Each Promela statement is either
  - executable:
  - Blocked
- There are six types of statement
  - Assignment: always executable
    - Ex. **`x=3+x`**, **`x=run A()`**
  - Print: always executable
    - Ex. **`printf(“Process %d is created.\n”, pid);`**
  - Assertion: always executable
    - Ex. **`assert( x + y == z)`**
  - Expression: depends on its value
    - Ex. **`x+3>0`**, **`0`**, **`1`**, **`2`**
    - Ex. **`skip, true`**
  - Send: depends on buffer status
    - Ex. **`ch1!m`** is executable only if **`ch1`** is not full
  - Receive: depends on buffer status
    - Ex. **`ch1?m`** is executable only if **`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

Program Execution Control

- 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

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

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

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

6 Types of Basic Statements

- Assignment: always executable
  - Ex. **`x=3+x`**, **`x=run A()`**
- Print: always executable
  - Ex. **`printf("Process %d is created.\n",_pid);`**
- Assertion: always executable
  - Ex. **`assert( x + y == z)`**
- Expression: depends on its value
  - Ex. **`x+3>0`**, **`0`**, **`1`**, **`2`**
  - Ex. **`skip, true`**
- Send: depends on buffer status
  - Ex. **`ch1!m`** is executable only if **`ch1`** is not full
- Receive: depends on buffer status
  - Ex. **`ch1?m`** is executable only if **`ch1`** is not empty

Critical Section Example

```
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);
}
```

```
[root@moonzoo spin_test]# ls
crit.pml
[root@moonzoo spin_test]# spin -a crit.pml
[root@moonzoo spin_test]# ls
crit.pml  pan.b  pan.c  pan.h  pan.m  pan.t
[root@moonzoo spin_test]# gcc pan.c
[root@moonzoo spin_test]# a.out
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)
[root@moonzoo spin_test]# ls
a.out  crit.pml  crit.pml.trail  pan.b  pan.c  pan.h
pan.m  pan.t
```

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
```

Revised 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 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)
```

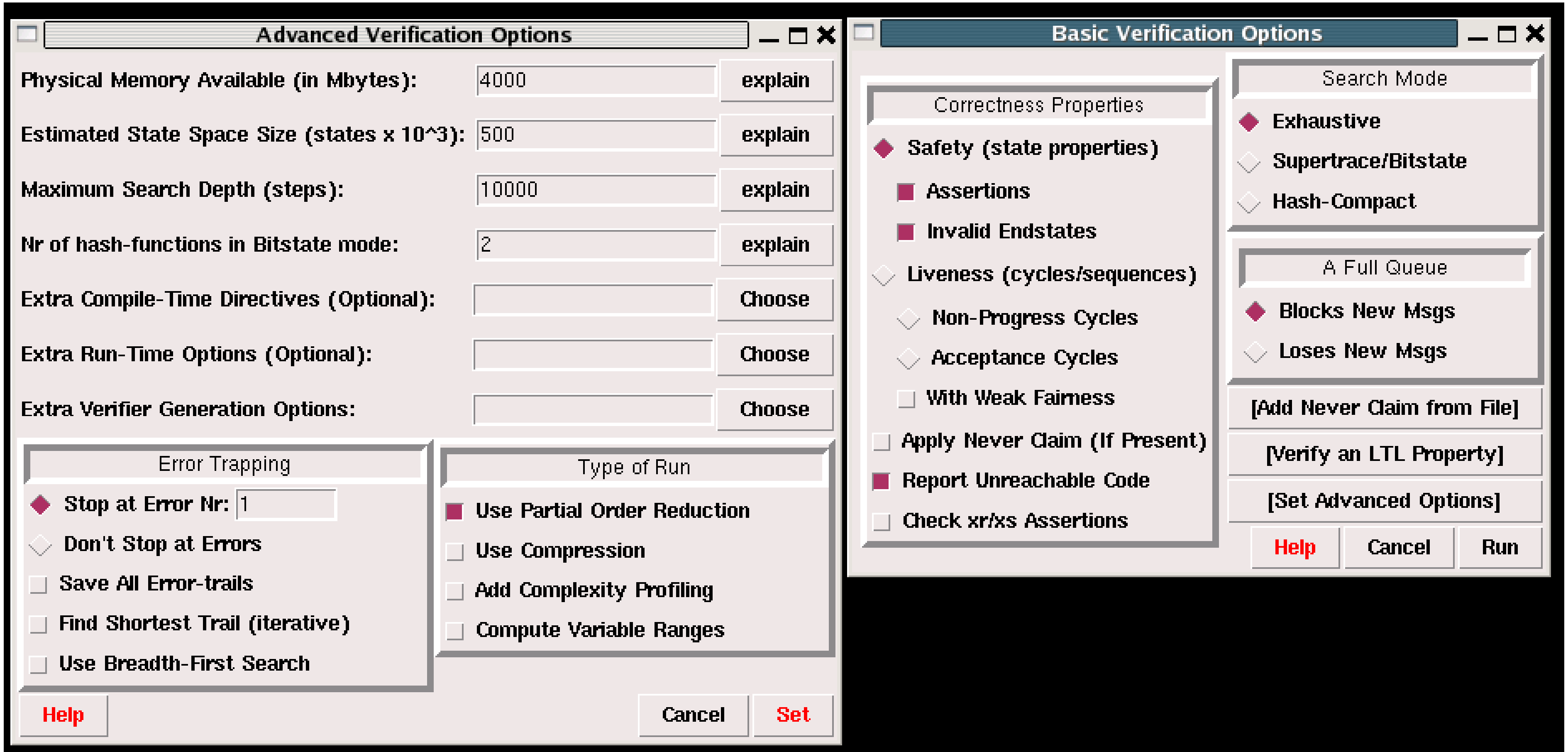
KAIST

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
```

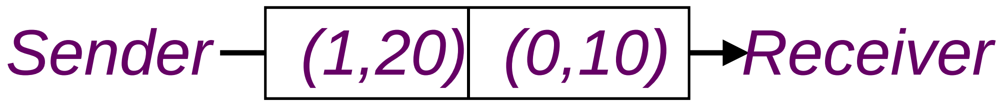
Options in XSPIN

- Now you have learned all necessary techniques to verify common problems in the SW development



Communication Using Message Channels

- Spin provides communications through various types of message channels
  - Buffered or non-buffered (rendezvous comm.)
  - Various message types
  - Various message handling operators
- Syntax
  - chan ch1 = [2] of { bit, byte};
    - ch1!0,10;ch1!1,20
    - ch1?b,bt;ch1?1,bt



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

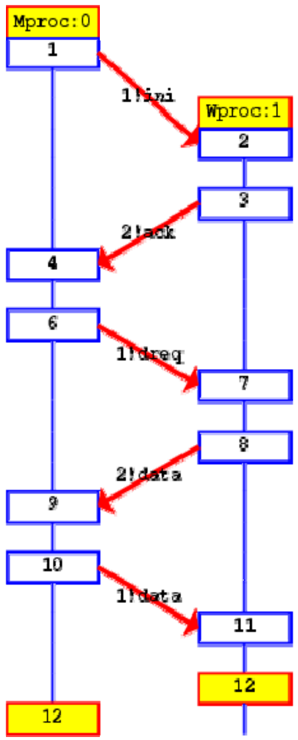
Operations on Channels

- **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

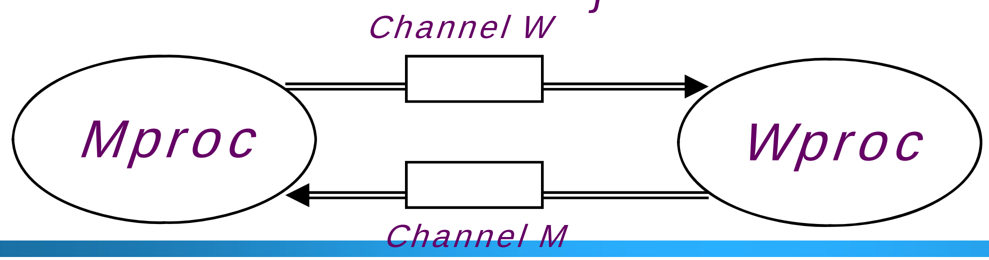
Faulty Data Transfer Protocol

(pg 27, data switch model proposed at 1981 at Bell labs)

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

active proctype Mproc()
{
        W!ini;    /* 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?ini;            /* 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 (pg 326)

```
/*
   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
}
```