Introduction to SPIN

Ralf Huuck

Acknowledgments

Parts of the slides are based on an earlier lecture by Radu Iosif, Verimag.

PROMELA (PROcess MEta LAnguage) is:
- a language to describe concurrent (distributed) systems:
  - network protocols, telephone systems
  - multi-threaded (-process) programs
- similar with some structured languages (e.g. C, Pascal)

SPIN (Simple Promela INterpreter) is a tool for:
- detecting logical errors in the design of systems e.g.:
  - deadlocks
  - assertions (e.g. race conditions)
  - temporal logic formulas

Features

Given a PROMELA model (program), SPIN can do:
- a random simulation i.e. it interprets the program
  - this mode cannot be used in exhaustive verification
  - useful for viewing error traces
- an exhaustive analysis
  - considers all possible executions of the program
  - finds all potential errors, both deadlock and user-specified

Deadlock:
- situation in which at least one process remains blocked forever while waiting for an inexistent event to happen

User specified constraints:
- assertions
- temporal (never) claims
PROMELA “Hello World”

```c
init {
    printf("Hello World\n");
}
```

The simplest erroneous specification

```c
init {
    0;
}
```

SPIN Simulation

```c
SPIN
C:\> spin –p hello.prom
simulation trace
```

SPIN Analysis (1)

```c
SPIN
C:\> spin –a dlock.prom
```

SPIN Analysis (2)

```c
SPIN
C:\> spin –p -t dlock.prom
```

GCC
The PROMELA language

Similar to C

- all C pre-processor directives can be used
  
  ```
  #define NAME 5
  ```
- definitions of types, variables, processes
- if, do, break, goto control flow constructs

Basic Types and Ranges

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit, bool</td>
<td>0..1</td>
</tr>
<tr>
<td>byte</td>
<td>0..255</td>
</tr>
<tr>
<td>short</td>
<td>-2^15 - 1 .. 2^15 - 1</td>
</tr>
<tr>
<td>int</td>
<td>-2^31 - 1 .. 2^31 - 1</td>
</tr>
</tbody>
</table>

**Warning:** type ranges are OS-dependent (just like in C)

At most one enumeration type

```pascal
mtype = {one, two, three};
```

Recode Types (user defined)

```pascal
typedef S {
    short a, b;
    byte x;
};
```

- look like C structures
Variables (same C syntax)

```c
int x, y;
int z = 0;
stype m = one;
```

Procedures

there are no procedures, only processes

```c
proctype foo(int x, y; bit b){...}
```

- the init process (like main in C)
  - has no parameters
  - can be activated by

```c
run foo(254, 255,0);
```

```c
active proctype foo(int x, y; bit b){...}
```

Scoping

- variables are global if declared outside any process
- variables are local a process if declared within its proctype declaration
- local variables shadow globals with the same name

Arrays and Constants

- arrays declared like variables:

```c
int vector[32];
```
- indexes are zero-based
- scoping rules apply (there might be global and local vectors)
Example

```c
#define length 64
mtype = {red, yellow, green};
byte state = green;
int counter;
bit memory[length];
init {
...
}
```

Expressions

- logical: ||, &&, !
- arithmetic: +, -, /, %
- relational: >, <, <=, >=, ==, !=
- vector access: v[i]
- record access: x.f
- process creation: run X()

Expressions

- are execution steps of processes
- an important characteristic is executability:

For instance:

```c
x <= 10;
```

is the (proper) way of expressing something like:

```c
wait until(x <= 10);
```

Statements (1)

- are execution steps of processes
- an important characteristic is executability:

For instance:

```c
x <= 10;
```

is the (proper) way of expressing something like:

```c
wait until(x <= 10);
```

Statements (2)

- Expression statements
  - not executable iff expression evaluates to 0
- Assignment statements
  - always executable
- Skip statements
  - always executable
    - do 'nothing' (only change control location)
- Print statements
  - always executable

Expressions

- logical: ||, &&, !
- arithmetic: +, -, /, %
- relational: >, <, <=, >=, ==, !=
- vector access: v[i]
- record access: x.f
- process creation: run X()
Statements (3)

- Assert statements
  ```
  assert( <expression> );
  ```
  - always executable
  - expression evaluates to zero => program exits

- Statements are atomic
  - in a concurrent program, each statement is executed without interleaving with other processes

Control Flow (1)

- Select construct
  ```
  if
    :: <choice1> -> <stat11>; <stat12>; ...
    :: <choice2> -> <stat21>; <stat22>; ...
    ...
  fi;
  ```
  - What does it do?
    - if a (random) choice is executable, continues execution with the corresponding branch
    - if no choice is executable, the whole select is not executable
    - if more than one choice is executable, we say the selection is non-deterministic

Control Flow (2)

- Loop construct
  ```
  do
    :: <choice1> -> <stat11>; <stat12>; ...
    :: <choice2> -> <stat21>; <stat22>; ...
    ...
  od;
  ```
  - What does it do?
    - same as the selection, except that at the end of a branch it loops back and repeats the choice selection

Control Flow (3)

- The else choice
  - is executable only when no other choice is executable

- The break statement
  - transfers control at the end of loop

- The goto statement
  - transfers control to a labeled location
Traffic Light Example

```cpp
mtype = {red, yellow, green};
byte state = green;
init {
    do
        :: (state == green) -> state = yellow;
        :: (state == yellow) -> state = red;
        :: (state == red) -> state = green;
    od
}
```

Concurrency

- Processes can spawn other processes using the `run` expression

```cpp
proctype foo(int x; byte y) {...}

init {
    int pid;
    pid = run foo(256, 255);
    /* or simply: */
    run foo(256, 255);
    ...
}
```

Concurrency

- Premises:
  - two or more processes composed of atomic statements
  - one processor shared between processes

- Problems:
  - worst-case complexity is exponential in number of processes
  - improper mutex (locking) may cause race conditions

Interleaving

- how many states can be reached in this example?
- express this number as function of:
  - \( K \) = number of processes
  - \( N \) = number of states/process
Reducing Complexity (1)

- if a statement inside atomic is not executable, transfer temporarily control to another process

Reducing Complexity (2)

- if a statement inside d_step is not executable => error (block inside d_step)
- no if, do, break, goto, run allowed inside d_step (i.e., deterministic step)

Apprentice Example

- **Good** apprentice
  ```cpp
  int counter = 0;
  active[2] proctype incr() {
    counter = counter + 1;
  }
  ```
  - creates 2 identical copies,
  - share variable `counter`
  - interleaving

- **Bad** apprentice
  ```cpp
  int counter = 0;
  active[2] proctype incr() {
    int tmp;
    tmp = counter + 1;
    counter = tmp;
  }
  ```
Apprentice Example

- **Good apprentice**
  ```c
  int counter = 0;
  active[2] proctype incr() {
    counter = counter + 1;
  }
  ```

- **Bad apprentice**
  ```c
  int counter = 0;
  active[2] proctype incr() {
    int tmp;
    tmp = counter + 1;
    counter = tmp;
  }
  ```

Mutual Exclusion (bad) Example

```c
proctype Y() {
  x = 1;
  y = 0;
  mutex ++;
  mutex --;
  y = 0;
}
```

```c
proctype X() {
  x = 1;
  y = 0;
  mutex ++;
  mutex --;
  y = 0;
}
```

Dekker's Mutual Exclusion

```c
proctype A() {
  x = 1;
  turn = Bturn;
  (y == 0) || (turn == Aturn);
  mutex ++;
  mutex --;
  x = 0;
}
```

```c
proctype B() {
  y = 1;
  turn = Aturn;
  (x == 0) || (turn == Bturn);
  mutex ++;
  mutex --;
  y = 0;
}
```

```c
proctype monitor() {
  assert(mutex != 2);
}
```

Bakery Mutual Exclusion

```c
proctype A() {
  do
  od
  if 1 => turnA = 1;
  turnA = turnA + 1;
  (turnB == 0) || (turnA < turnB);
  mutex ++;
  mutex --;
  turnA = 0;
}
```
Communication

1. Message passing
2. Rendez-vous synchronization

- Both methods rely on channels:
  
  ```
  chan q = [<dim>] of {<type>};
  chan q; /* just declaration */
  ```

  - if `dim` is 0 then `q` is a rendez-vous port
  - otherwise `q` is an **asynchronous** channel

Communication (2)

- Send statement
  
  ```
  q ! <expr>;
  ```

  - for rendez-vous channels is executable iff another process is ready to receive at the same time  both processes are involved in a rendez-vous
  - for asynchronous queues is executable iff the queue is not full

Communication (3)

- Receive statement
  
  ```
  q ? <var>;
  q ? _;
  ```

  - for rendez-vous channels is executable iff another process is ready to send at the same time  both processes are involved in a rendez-vous
  - for asynchronous queues is executable iff the queue is not empty
chan lock = [1] of Bit;
proc type foo(chan q) {
    q ? 1;
    /* critical section */
    q ! 1;
}
init {
    lock ! 1;
    run foo(lock);
    run foo(lock);
}

Inter-locking Example

Correctness Claims

Basic Assertions

 assertions
 end state labels
 progress state labels
 accept state labels
 never claims
 trace assertions
 all provide means to check requirements

assert (expression)

SPIN reports any violation of assertions during simulation or verification.

spin: line 2 "pan", Error: assertion violated
Meta Labels

- labels with special meaning
- reserved
- only available in verification mode

Labels for
- end states
- progress states
- accept states

End States

- an end state in PROMELA is a state with no successors
- end states can be labeled as valid by introducing labels that start with `end`
  - `end0`, `endsome`, ...
- at the end of each process there is a default valid end state
- a deadlock is a state with no successors in which at least one process is not in a valid end state

Example End State

```
proctype Y() {
    x = 1;
    endY:
    y == 0;
    mutex ++;
    mutex --;
    x = 0;
}
```

```
proctype X() {
    y = 1;
    endX:
    x == 0;
    mutex ++;
    mutex --;
    y = 0;
}
```

Allow processes to block.

Progress States

- similar to end states
- progress states are introduced by labels that start with `progress`
  - `progress0`, `progresssome`, ...
- verifier must in each execution reach progress label infinitely often
- typically requires fairness
Example Progress State

```c
proctype X() {
  x = 1;
y == 0;
mutex ++;
mutex --;
progressX:
y = 0;
}
proctype Y() {
  x = 1;
y == 0;
mutex ++;
mutex --;
progressY:
x = 0;
}
```

Mutual exclusion infinitely many times for both processes.

Accept States

- dual to end states
- accept states are introduced by labels that start with `accept`
  - `accept0, accept1, ...`
- verifier must not reach any execution reach accept label in infinitely often

Example Accept State

```c
proctype Y() {
  x = 1;
y == 0;
mutex ++;
mutex --;
acceptY:
x = 0;
}
proctype X() {
  x = 1;
y == 0;
mutex ++;
mutex --;
progressX:
y = 0;
}
```

Mutual exclusion infinitely many times for X only.

Never Claims (1)

- special type of process, is instantiated once
- used to detect illegal behaviors
- SPIN has LTL to never claim translator

```c
never (statement)
```
SPIN reports any violation of never claims during simulation or verification.

```
never { 
  do 
  ;; if p -> break
  ;; else
  od 
}
```

Ralf Huuck COMP 4152

SPIN Options

- Simulation
  - random simulation: spin model
  - i performs interactive simulation
  - jN skips first N steps of random/guided sim.
  - p shows execution of states
  - s/r shows details about send/receive
  - t interactive sim following on produced execution trace
  - v verbose
  - and many more!

- Generating Verifier
  - -a syntax check and verifier generation
  - -f formula generates never claim from LTL
  - -F file same but from file
Verification

- A suppresses basic assertion violations
- a use for accept cycle detection
- f uses weak fairness
- 1 use for progress cycles

References

- http://spinroot.com/
- quick references http://spinroot.com/spin/Man/Quick.html

You better read them 😊