Formal Methods, CS 6110, Fall 2007 Notes for Week-1, 8/23/07

These notes will teach you Promela and SPIN through many examples.

1 ex1: Warmup

- Pretend that this was an attempt by somebody to generate states which contain even numbered values of `x`. Of course, that is not what happens: states where `x` has odd values are also generated.
- The total number of states stored, as well as the number of transitions reported by SPIN are very low. All this is fine. It is optimizing things, as there are no properties being checked.

```
byte x; /* initialized to 0 by default */
proctype test()
{do
 :: x++ ; x++
 od
}
init {
 run test();
}
```

2 ex2: Illustrates another way to init procs

The use of `active` is handy, especially if all processes (here there is only one) are started in a similar manner. See details in the SPIN book.

```
byte x;
active proctype test()
{do
 :: x++ ; x++
 od
}
```

3 ex3: Property specification

- We are pretending that the designer meant for only even values of `x` to appear. Keeping that in mind, he wrote "It is bad to see x==3."
- In Promela, the right curly brace of the `never` automaton is a Büchi acceptance state with a self-loop labeled with `true` to itself. Keeping this in mind, the *synchronous product* of the the `never` automaton and the user's process automaton is performed. The language of this product machine is non-empty. Therefore, there is an error: meaning `x==3` indeed happens! See my book for additional explanations.
- The number of states/transitions goes up, as a property machine is (finally!) observing `x`.

```
byte x;
active proctype test()
{do
 :: x++ ; x++
 od
}
never {
do
:: skip
:: x==3 -> break
od
}
```

4 ex4: What the user meant might be atomic increases by two

Here, the user got his encoding of "even jumps" right.

```
byte x;
active proctype test()
{do
 :: atomic { x++ ; x++ }
 od
}
never {
do
:: skip
:: x==3 -> break
od
}
```

5 ex5: Interleave another proc

We interleave another process. Only the number of transitions go up due to back-edges. The number of states do not go up from the previous experiments because we had already reached the maximum number of different states of `byte x`.

```
/* Num of states stay the same,
   # transitions go up wrt ex4.pr */

byte x;
active proctype p1()
{do
 :: atomic { x++ ; x++ }
 od
}
active proctype p2()
{do
 :: atomic { x-- ; x-- }
 od
}
never {
do
:: skip
:: x==3 -> break
od
}
```

6 ex6: Interleave proctypes that use different variables

State explosion kicks in. The search got truncated, but already we see the number of states shooting up.

```
/* Smart reductions saved states */
byte x,y;
active proctype p1()
{do
 :: atomic { x++ ; x++ }
 od
}
active proctype p2()
{do
 :: atomic { y++ ; y++ }
 od
}
never {
do
```

```
:: skip
:: x==3 -> break
od
}
```

7 ex8: Truncation of DFS

Verify property that is expected to be false.

```
/* state explosion - DFS got truncated */
/* ex8 reruns with higher depth */

byte x,y;
active proctype p1()
{do
 :: atomic { x++ ; x++ }
 od
}
active proctype p2()
{do
 :: atomic { y++ ; y++ }
 od
}
never {
do
:: skip
:: (x==3)&&(y==233) -> break
od
}
```

8 ex9: Another variation

Here the property is violated.

```
/* Violation found */

byte x,y;
active proctype p1()
{do
 :: atomic { x++ ; x++ }
 od
}
active proctype p2()
{do
 :: atomic { y++ ; y++ }
 od
}
never {
do
:: skip
:: (x==2)&&(y==232) -> break
od
}
```

9 ex10: Use Local Vars

more modular coding and also partial-order reduction (technique to contain state explosion) works much better aided by scope info.

```
/* Local variables */
```

```
byte pid1, pid2;

proctype p1()
{byte x; /* Also init to 0 */
 do
 :: atomic { x++ ; x++ }
 od
}
proctype p2()
{byte y; /* Also init to 0 */
 do
 :: atomic { y++ ; y++ }
 od
}
init {
 atomic {
 pid1 = run p1();
 pid2 = run p2();
 }
}

never {
do
:: skip
:: (p1:x==2)&&(p2:y==232) -> break
od
}
```

10 ex11: Introducing channels

No violations detected, as the rendezvous channel ensures zero "slack."

```
/* Channels */

chan ch = [0] of { byte }; /* rendezvous channel */

active proctype p1()
{byte x; /* local var x init to 0 */
 do
 :: x++ -> ch!x /* ; and -> are the same */
 od
}
active proctype p2()
{byte y,z; /* can be named x, but keeping distinct names */
 do
 :: ch?y -> z++ /* z tracks value of x */
 od
}
never {
do
:: skip
:: (p2:y - p2:z) > 1 -> break
od
}
```

11 ex12: More channels - READ ALL WARNINGS!

- This should produce a violation.
- It didn't!!
- However, you now wake up and pay attention to the xspin console which says "remote references not compatible with POR." XSPIN keeps POR on, by default.

- Turn off POR, and a violation is produced!

```
/* Channels */

chan ch = [0] of { byte }; /* rendezvous channel */

active proctype p1()
{byte x; /* local var x init to 0 */
 do
 :: x++ -> ch!x /* ; and -> are the same */
 od
}
active proctype p2()
{byte y,z; /* can be named x, but keeping distinct names */
 do
 :: ch?y -> z++ /* z tracks value of x */
 od
}
never {
do
:: skip
:: (p1:x - p2:z) > 1 -> break
od
}
```

12 ex13: depth-bounding for shorter error traces

- Turn off POR
- Bug caught; but MSC too long
- Depth-bound search and all is fine

```
/* set depth-bound of DFS to around 20 */

chan ch = [1] of { byte }; /* buffering (non-rendezvous) channel */

active proctype p1()
{byte x; /* local var x init to 0 */
 do
 :: x++ -> ch!x /* ; and -> are the same */
 od
}
active proctype p2()
{byte y,z; /* can be named x, but keeping distinct names */
 do
 :: ch?y -> z++ /* z tracks value of x */
 od
}
never {
do
:: skip
:: (p1:x - p2:z) > 2 -> break
od
}
```

13 ex14: A Vanilla Büchi Check

- `x = x + 3` is atomic in Promela
- Anyway, depth-bound search; else the search goes off on the "DFS tangent."
- Violation trace shown in comments after `never`.

```
/* Good for understanding Buchi */
/* Depth bound for shorter traces - 1000 seems necessary */
/* Must be before exp tree to the left of "magic trace" needed */
/* Rearranging p1 and p2 might help */

active proctype p1()
{byte x;
 do
 :: x = x + 3 /* this statement is atomic, unlike in C !! */
 od
}
active proctype p2()
{byte y;
 do
 :: y = y + 5
 od
}
never {
do
:: skip
:: (p1:x == p2:y) -> break
od;
accept: goto accept; /* not needed but looks Buchi */
}

/*
p1 ; never; p2 ; never;
p1 ; never; p2 ; never;
p1 ; never; p2 ; never;
p1 ; never;
p1 ; never

NOW a Buchi acceptance loop will form.

Thus L(S) not contained in L(P),
because L(S) intersect complement(L(P))
is non-empty.

*/
```

14 phil3ll.pr

The inevitable “Naive Philosophers” code. This reveals a livelock.

```
mtype = {are_you_free, yes, no, release}
byte progress; /* SPIN initializes all variables to 0 */
proctype phil(chan lf, rf; int philno)
{ do
  :: do
     :: lf!are_you_free ->
        if
        :: lf?yes -> break
        :: lf?no
        fi
     od;
     do
     :: rf!are_you_free ->
        if
        :: rf?yes -> progress = 1 -> progress = 0
                  -> lf!release   -> rf!release -> break
        :: rf?no  -> lf!release   ->  break
        fi
     od
  od
}
```

```
proctype fork(chan lp, rp)
{ do
  :: rp?are_you_free -> rp!yes ->
     do
     :: lp?are_you_free -> lp!no
     :: rp?release      -> break
     od
  :: lp?are_you_free -> lp!yes ->
     do
     :: rp?are_you_free -> rp!no
     :: lp?release      -> break
     od
  od
}
init {
   chan c0 = [0] of { mtype }; chan c1 = [0] of { mtype };
   chan c2 = [0] of { mtype }; chan c3 = [0] of { mtype };
   chan c4 = [0] of { mtype }; chan c5 = [0] of { mtype };
   atomic {
     run phil(c5, c0, 0); run fork(c0, c1);
     run phil(c1, c2, 1); run fork(c2, c3);
     run phil(c3, c4, 2); run fork(c4, c5); }
}
never { /* Negation of []<> progress */
 do
 :: skip
 :: (!progress) -> goto accept;
 od;
 accept: (!progress) -> goto accept;
}
```

Develop a version of the Philosophers protocol that is deadlock-free and guarantees communal progress.

15 phil3ll.pr written using Progress labels

I don't use Progress labels much - but here is the idea, in case you want to encode the previous using progress labels.

```
mtype = {are_you_free, yes, no, release}

proctype phil(chan lf, rf; int philno)
{ do
  :: do
     :: lf!are_you_free ->
        if
        :: lf?yes -> break
        :: lf?no
        fi
     od;
     do
     :: rf!are_you_free ->
        if
        :: rf?yes /* eat */
                  -> progress: lf!release   -> rf!release -> break
        :: rf?no  -> lf!release   ->  break
        fi
     od
  od
}
proctype fork(chan lp, rp)
{ do
  :: rp?are_you_free -> rp!yes ->
     do
     :: lp?are_you_free -> lp!no
```

```
     :: rp?release      -> break
     od
  :: lp?are_you_free -> lp!yes ->
     do
     :: rp?are_you_free -> rp!no
     :: lp?release      -> break
     od
  od
}
init {
   chan c0 = [0] of { mtype }; chan c1 = [0] of { mtype };
   chan c2 = [0] of { mtype }; chan c3 = [0] of { mtype };
   chan c4 = [0] of { mtype }; chan c5 = [0] of { mtype };
   atomic {
     run phil(c5, c0, 0); run fork(c0, c1);
     run phil(c1, c2, 1); run fork(c2, c3);
     run phil(c3, c4, 2); run fork(c4, c5); }
}
```