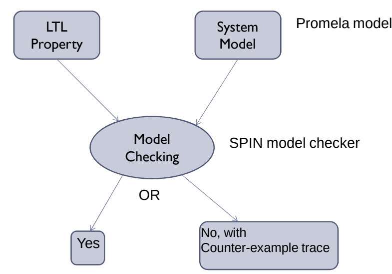
SPIN Model Checker
CS 4271

Abhik Roychoudhury
National University of Singapore

From last lecture



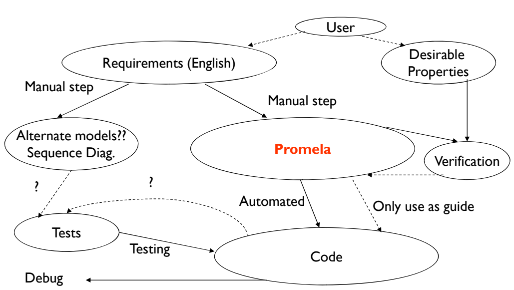
SPIN

- A tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A model checker
    - The nested DFS algorithm discussed in last class is implemented inside the SPIN model checker!
  - A random simulator for system simulation
  - Promela models can be automatically generated from a safe subset of C.

Our Usage

- Learn Promela, a low-level modeling language.
- Use it to model simple concurrent system protocols and interactions.
- Gain experience in verifying such concurrent software using the SPIN model checker.
- Gives a feel (at a small scale)
  - What are hard-to-find errors ?
  - How to find the bug in the code, once model checking has produced a counter-example ?

Our Usage



Verification vs. testing

- Sequential Program
  - Test
    - Try one/selected inputs
  - Verify
    - Try all possible inputs and check whether output is as "expected"
    - Need to specify "expectation"
- Concurrent Program
  - Test
    - Try one/selected inputs and/or thread schedules.
  - Verify
    - Try all possible inputs and for them all possible schedules
    - No notion of output, specify "expectation" as temp. logic properties over exec. trace

Why Promela ?

- Extensive support of various control constructs for computation.
  - Assignments, Assert, If, Do
  - Ideas from guarded command languages
- Dynamic creation of processes supported.
  - Gives the flavor of a realistic multi-threaded programming language
  - Yet supported directly by a model checker !!
  - Ideal for our purposes in this course.

Features of Promela

- Concurrency
  - Multiple processes in a system description.
- Asynchronous Composition
  - At any point one of the processes active.
  - Interleaving semantics
- Communication
  - Shared variables
  - Message passing
    - Handshake (synchronous message passing)
    - Buffers (asynchronous message passing)

Features of Promela

- Within a process
  - Non-determinism : supports the situation where all details of a process may not be captured in Promela model.
  - Standard C-like syntax
    - Assignment
    - Switch statement
    - While loop
    - Guarded command
      - Guard and body may not evaluated together, that is, atomically.

Example 1

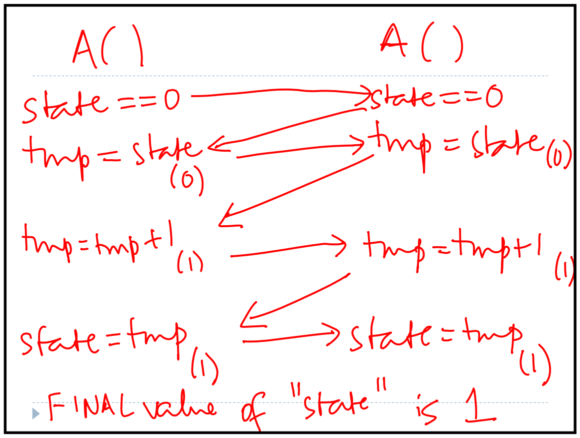
```
byte state = 0;

proctype A()
{  byte tmp;

    (state==0) -> tmp = state;
     tmp = tmp+1;
     state = tmp;
}

init  { run A() ; run A(); }
```

We need to define how processes are scheduled.



Example 2

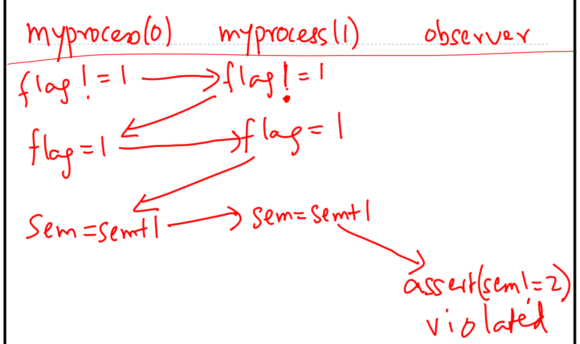
```
bit  flag;
byte sem;
proctype myprocess(bit i)
{    (flag != 1) ->flag = 1;
     sem = sem + 1;
     sem = sem – 1;
     flag =  0;
}
proctype observer() {
     assert( sem != 2 );
}
```

```
init {
     atomic{
          run myprocess(0));
          run myprocess(1)
          run observer();
     }
}
```

All three processes Instantiated together

Concepts in Example 2

- Interleaved execution among threads.
- Shared variable communication
- Unintended shared variable values
  - Due to unforeseen interleavings
- And, a mechanism for
  - Specifying the unintended behavior
  - Producing the interleaving that produces this unintended behavior.
  - We only do this in our modeling environment – hard to do this for real programs!



SPIN's process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the "active" processes will execute (non-deterministic choice here)
  - A process is active, if it has been created, and its "next" statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
    - We have not seen such an example yet !

Will this loop terminate?

Non-determinism within a single process.

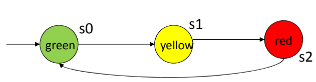
```
byte count;

        proctype counter()
        {
                do
                :: count = count + 1
                :: count = count - 1
                :: (count == 0) -> break
                od;
        }
```

Enumerate the reasons for non-termination in this example

This loop will not terminate

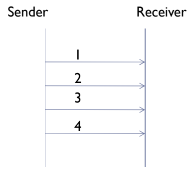
```
active proctype TrafficLightController() {
     byte color = green;
     do
      :: (color == green) -> color = yellow;
      :: (color == yellow) -> color = red;
      :: (color == red) -> color = green;
      od;
}
```



Channels

- SPIN processes can communicate by exchanging messages across channels
  - Apart from communication via shared variables.
- Channels are typed.
- Any channel is a FIFO buffer.
- Handshakes supported when buffer is null.
- chan ch = [2] of bit;
  - A buffer of length 2, each element is a bit.
- Array of channels also possible.
  - Talking to diff. processes via dedicated channels.

Handshake or not?



Handshake communication
<!1, ?1>, <!2, ?2>, <!3, ?3>, …
(only possible interleaving)
Buffer of length 2
!1, !2, ?1, !3, ?2, …
(also possible)

What is the minimum sized buffer needed to allow this interleaving?

!1, !2, ?1, ?2, !3, !4, ?3, ?4, …

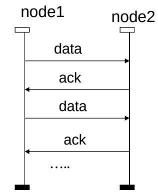
Example with channels

```
chan data, ack = [1] of bit;

proctype node1() {
do
:: data!1;
:: ack?1;
od
}

proctype node2() {
  do
  :: ack!1;
  :: data?1;
  od
}

init{ atomic{
      run node1(); run node2();
   }
}
```



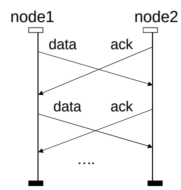
Example with channels

```
chan data, ack = [1] of bit;

proctype node1() {
do
:: data!1;
:: ack?1;
od
}

proctype node2() {
  do
  :: ack!1;
  :: data?1;
  od
}

init{ atomic{
      run node1(); run node2();
   }
}
```



SPIN Execution Semantics

- Select an enabled transition of any thread, and execute it.
- A transition corresponds to one statement in a thread.
  - Handshakes must be executed together.
    - *chan x = [0] of {…};*
    - *x!1 || x?data*

SPIN Execution Engine

```
while ( (E = executable(s)) != {})
     for some (p,t) ∈ E
     {   s' = apply(t.effect, s);   /* execute the chosen statement */
         if (handshake == 0)
         {       s = s' ;
                 p.curstate = t.target
         }
         else{ …
```


SPIN Execution Engine

```
                /* try to complete the handshake */
                E' = executable(s'); /* E' ={} ⇒ s unchanged */
                for some (p', t') ∈ E'
                {      s = apply(t'.effect, s');
                       p.curstate = t.target;
                       p'.curstate = t'.target;
                }
                handshake = 0
         }  /*  else  */
      } /*  for some (p, t) ∈ E */
   }  /* while ((E = executable(s))  … */
   while  (stutter) { s = s }
```


Model Checking in SPIN

- (P1 || P2 || P3) $\models \varphi$
  - P1, P2, P3 are Promela processes
  - φ is a LTL formula
- Construct a state machine via
  - M, asynchronous composition of processes P1, P2, P3
  - $A_{\neg\varphi}$, representing $\neg\varphi$
- Show that "language" of $M \times A_{\neg\varphi}$ is empty
  - No accepting cycles.
- All these steps have been studied by us !!

Specifying properties in SPIN

- Invariants
  - Local: via assert statement insertion
  - Global: assert statement in a monitor process
- Deadlocks
- Arbitrary Temporal Properties (entered by user)
  - SPIN is a LTL model checker.
  - LTL properties can be entered as input to the checker!
    - Shown in the lab hour of the last lecture!

Connect system & property in SPIN

- **System model**

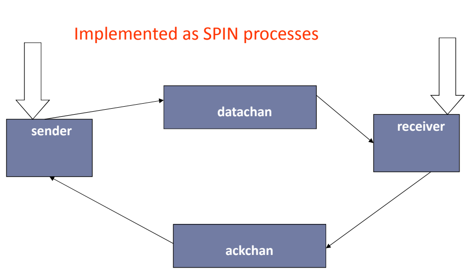
```
int  x = 100;
active proctype A()
{  do
    :: x %2 -> x = 3*x+1
    od
}
active proctype B()
{  do
    :: !(x%2) -> x = x/2
    od
}
```

- **Property**
  - GF (x = 1)
- Insert into code
  - #define q (x == 1)
- Now try to verify GF q

More Involved Example

- Alternating Bit Protocol
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.

ABP Architecture



Sender & Receiver code

- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

```
active proctype Sender()
{   bit out, in;
    do
    :: datachan!out ->
            ackchan?in;
            if
            :: in == out -> out = 1- out;
            :: else fi
    od
}
```

```
active proctype Receiver()
{   bit in ;
    do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
    od
}
```


Timeouts

- Special feature of the language
  - Time independent feature.
    - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Model Checking in SPIN

- SPIN performs model checking by Nested DFS
  - Discussed in the past lecture !!

- Find acceptance states reachable from initial states (DFS).
- Find all such acceptance states which are reachable from itself (DFS).
- Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.

Some Common Questions

- How does the product of the system and property automata work ?
  - How is the interaction between system and property automata achieved ?
- Can we specify LTL properties directly ?
  - Yes, you can do so in SPIN.
- Can we model/verify pgms with procedures
  - Yes.

Let us finish with a real-life situation

- July 4, 1997
  - NASA's Pathfinder landed on Mars.
  - Tremendous engineering feat.
  - Hard to design the control software with concurrency and priority driven scheduling of threads.
  - The SpaceRover would lose contact with earth in unpredictable moments.

The Mars Pathfinder problem

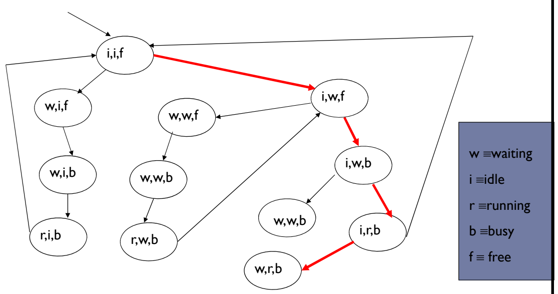
"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"." …

Essence of the problem in SPIN

```
mtype = { free, busy, idle, waiting, running };
mtype H = idle;   mtype L = idle; mtype mutex = free;
```

active proctype high();	active proctype low() provided (H == idle)
{end: do :: H = waiting; atomic { mutex == free -> mutex = busy }; H = running; atomic{ H=idle; mutex=free } od }	{ end: do :: L = waiting; atomic{ mutex== free-> mutex = busy}; L = running; atomic{ L=idle; mutex = free } od }

State Space Graph



Source of deadlock

- Counterexample
  - Low priority thread acquires lock
  - High priority thread starts
  - Low priority process cannot be scheduled
  - High priority thread blocked on lock
- Actual error was a bit more complex with three threads of three different priorities
  - Timer went off with such a deadlock resulting in a system reset and loss of transmitted data.

More readings on SPIN

- http://spinroot.com/spin/Man/Manual.html
  - SPIN manual
- The model checker SPIN (Holzmann)
  - IEEE transactions on software engineering, 23(5), 1997.
- http://spinroot.com/spin/Doc/SpinTutorial.pdf
  - SPIN beginner's tutorial (Theo Ruys)
- ``The SPIN model checker: primer and reference manual", by Holzmann (mostly chapters 2,3,7,8)
  - *This one is optional reading.*

Exercise 1 – Model Checking

Two Computer Engineering students are taking the CS4271 exam. We must ensure that they cannot leave the exam hall at the same time. To prevent this, each student reads a shared token *n* before leaving the hall. The shared token is an arbitrary natural number. The global state of the system is given by *s1, s2, n where s1 and s2 are the local states of students 1 and 2 respectively.* Note that $s1 \in \{in, out\}$, $s2 \in \{in, out\}$. *The pseudo-code executed by the two students are:*

```
do forever{
if s1 = in and n is odd
        { s1 := out }
else if s1 = out
     { s1 := in; n := 3*n+1}
else {do nothing }
}
```

```
do forever{
   if s2 = in and n is even
              {s2 := out}
   else if s2 = out and n is even
          { s2 := in ; n := n/2 }
   else { do nothing }
}
```

- The two student processes are executed asynchronously. Every time one process is scheduled, it *atomically executes one iteration of its loop. The above system is an infinite state system. Design a* finite state abstraction and draw the global automata for the abstracted system. Your abstraction should be refined enough to prove mutual exclusion. Initially s1 = in and s2 = in.
- Consider the mutual exclusion property. Specify it in LTL. Is the mutual exclusion property true in this case? Why or why not?

Exercise 2 – Concurrency, Model Checking

- Consider an asynchronous composition of two processes, i.e. in any time step only one of them makes a move. These processes communicate via a single shared variable x. Both processes are executing the following infinite loop:
- while true do x := x + x
- Every time one of the processes is scheduled, it atomically executes x := x + x and then again another process is scheduled. The initial value of x is . What will be the values of x reached during system execution and why ?

Exercise 2 – Take a guess!

Suppose the infinite loop is compiled by a naïve compiler as follows. The sequence of instructions executed by process A and process B are shown. The processes are running asynchronously, and each time a process is scheduled, only its next instruction is executed atomically. Initially x = 1.

	Process A		Process B
$Loop_A$:	$reg_A^1 = x$	$Loop_B$:	$reg_B^1 = x$
	$reg_A^2 = x$		$reg_B^2 = x$
	$reg_A^3 = reg_A^1 + reg_A^2$		$reg_B^3 = reg_B^1 + reg_B^2$
	$x = reg_A^3$		$x = reg_B^3$
	go to $Loop_A$		go to $Loop_B$

- What are all the possible values that x reach during system execution in this situation ? Explain your answer. Note that x is a shared global variable and reg_A^i, reg_B^i are local registers in processes A and B respectively.

