Model Checking and Testing

- Classical IPC Problem: Dining Philosophers
- Model Checking
  - Modeling Language: SMV
  - Specification Language: CTL
  - Presented by Jason Simas
- Testing
  - Implementation Language: Java
  - Presented by Evren Sahin
Dining Philosophers

• IPC Problem
  – Asynchronous processes & shared resources
    • Philosophers are processes
    • Resources are forks

• Solution: Modern Operating Systems pg. 127
  – Freedom from starvation/deadlock
  – Exclusive use of resources
  – Maximal usage of resources
#define N 5                /*number of philosophers*/
#define LEFT  (i+N-1)%N    /*i's left neighbour*/
#define RIGHT (i+1)%N      /*i's right neighbour*/
#define THINKING 0         /*philosopher is thinking*/
#define HUNGRY   1         /*philosopher is trying to get the forks*/
#define EATING   2         /*philosopher is eating*/

typedef int semaphore;     /*semaphores are special kind of integers*/
int state[N];              /*array to keep track of everyone's state*/
semaphore mutex;           /*mutual exclusion for critical regions (init 1)*/
semaphore s[N];           /*one semaphore per philosopher (init 0)*/

void philosopher(int i) {  /*i:philosopher number, from 0 to N-1*/
    while (TRUE) {          /*repeat forever*/
        think();               /*philosopher is thinking*/
        take_forks(i);         /*acquire two forks or block*/
        eat();                 /*yum-yum, spaghetti*/
        put_forks(i);          /*put both forks back on table*/
    }
}
Solution: Other

```c
void take_forks(int i) {
    /* i: philosopher number, from 0 to N-1 */
    down(&mutex);  /* enter critical region */
    state[i] = HUNGRY;  /* record fact that philosopher is hungry */
    test(i);  /* try to acquire 2 forks */
    up(&mutex);  /* exit critical region */
    down(&s[i]);  /* block if forks were not acquired */
}

void put_forks(int i) {
    /* i: philosopher number, from 0 to N-1 */
    down(&mutex);  /* enter critical region */
    state[i] = THINKING;  /* philosopher has finished eating */
    test (LEFT));  /* see if left neighbour can now eat */
    test (RIGHT);  /* see if right neighbour can now eat */
    up(&mutex);  /* exit critical region */
}

void test (int i) {
    /* i: philosopher number, from 0 to N-1 */
    if (state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
SMV : Overview

• Abstraction Level: pseudo instructions (137 loc)
• Model checked for 5, 4, and 3 philosophers
  – Checked both even and odd number of philosophers.
• Checked whether:
  – Each philosopher gets to eat infinitely often
    • No starvation, no deadlock
  – If a philosopher is eating, its neighbors are not eating
    • Exclusive use of resources
  – Possibility for non-neighbors to eat simultaneously
    • Maximal usage of resources
MODULE main
VAR
  ph_0 : process philosopher (0, n, mutex, sems, states);
  ph_1 : process philosopher (1, n, mutex, sems, states);
  ph_2 : process philosopher (2, n, mutex, sems, states);
  ph_3 : process philosopher (3, n, mutex, sems, states);
  ph_4 : process philosopher (4, n, mutex, sems, states);
  mutex : boolean;
  sems : array 0 .. 4 of boolean;
  states : array 0 .. 4 of {THINKING, HUNGRY, EATING};
ASSIGN
  init (mutex) := 1;
DEFINE
  n := 5;
SPEC --ppl eat (no starvation)
  (AG ((AF ph_0.eating) & (AF ph_1.eating) &
       (AF ph_2.eating) & (AF ph_3.eating) & (AF ph_4.eating)))
SPEC --ppl eat with forks (mutual exclusion)
  (AG ((ph_0.eating -> (!ph_4.eating & !ph_1.eating)) &
     (ph_1.eating -> (!ph_0.eating & !ph_2.eating)) &
     (ph_2.eating -> (!ph_1.eating & !ph_3.eating)) &
     (ph_3.eating -> (!ph_2.eating & !ph_4.eating)) &
     (ph_4.eating -> (!ph_3.eating & !ph_0.eating)))))
SPEC --ppl eat simultaneously
  (AG ((EF (ph_0.eating & ph_2.eating)) &
     (EF (ph_1.eating & ph_3.eating)) &
     (EF (ph_2.eating & ph_4.eating)) &
     (EF (ph_3.eating & ph_0.eating)) &
     (EF (ph_4.eating & ph_1.eating)))))
MODULE main
VAR
  ph_0 : process philosopher (0, n, mutex, sems, states);
  ph_1 : process philosopher (1, n, mutex, sems, states);
  ph_2 : process philosopher (2, n, mutex, sems, states);
  ph_3 : process philosopher (3, n, mutex, sems, states);
  mutex : boolean;
  sems : array 0 .. 3 of boolean;
  states : array 0 .. 3 of {THINKING, HUNGRY, EATING};
ASSIGN
  init (mutex) := 1;
DEFINE
  n := 4;
SPEC --ppl eat (no starvation)
  (AG ((AF ph_0.eating) & (AF ph_1.eating) & (AF ph_2.eating) & (AF ph_3.eating)))
SPEC --ppl eat with forks (mutual exclusion)
  (AG ((ph_0.eating -> (!ph_3.eating & !ph_1.eating)) &
        (ph_1.eating -> (!ph_0.eating & !ph_2.eating)) &
        (ph_2.eating -> (!ph_1.eating & !ph_3.eating)) &
        (ph_3.eating -> (!ph_2.eating & !ph_0.eating))))
SPEC --ppl eat simultaneously
  (AG ((EF (ph_0.eating & ph_2.eating)) &
        (EF (ph_1.eating & ph_3.eating)))))
MODULE philosopher (i, n, mutex, sems, states)
VAR
insns : {thinking_, take_forks_, eating_, put_forks_};
take_forks : {begin, down_mutex, state_hungry, if, state_eating, up_sem, up_mutex, down_sem, end};
put_forks : {begin, down_mutex, state_thinking, left_if,
    left_state_eating, left_up_sem, right_if,
    right_state_eating, right_up_sem, up_mutex, end};
ASSIGN
...
DEFINE
left := (i + n - 1) mod n;
right := (i + 1) mod n;
leftleft := (left + n - 1) mod n;
lefright := i;
rightleft := i;
rightright := (right + 1) mod n;
eating := (insns = eating_);
thinking := (insns = thinking_);
hungry := (insns = take_forks_);
FAIRNESS
(AG (AF (running & mutex)))
next (mutex) := case
    take_forks = down_mutex & mutex : 0;
    take_forks = up_mutex : 1;
    put_forks = down_mutex & mutex : 0;
    put_forks = up_mutex : 1;
    1 : mutex;
esac;
init (states[i]) := THINKING;
next (states[i]) := case
    take_forks = state_hungry : HUNGRY;
    take_forks = state_eating : EATING;
    put_forks = state_thinking : THINKING;
    1 : states[i];
esac;
init (sems[i]) := 0;
next (sems[i]) := case
    take_forks = up_sem : 1;
    take_forks = down_sem & sems[i] : 0;
    1 : sems[i];
esac;
init (take_forks) := begin;
next (take_forks) := case
    insns = take_forks_ &
    take_forks = begin : down_mutex;
    take_forks = down_mutex & mutex : state_hungry;
    take_forks = state_hungry : if;
    take_forks = if & (states[i] = HUNGRY &
        states[left] != EATING & states[right] != EATING) :
        state_eating;
    take_forks = if & !(states[i] = HUNGRY &
        states[left] != EATING & states[right] != EATING) :
        up_mutex;
    take_forks = state_eating : up_sem;
    take_forks = up_sem : up_mutex;
    take_forks = up_mutex : down_sem;
    take_forks = down_sem & sems[i] : end;
    take_forks = end : begin;
1 : take_forks;
esac;
SMV : Model : Philosopher : put_forks

init (put_forks) := begin;
next (put_forks) := case
  insns = put_forks &
  put_forks = begin : down_mutex;
  put_forks = down_mutex & mutex : state_thinking;
  put_forks = state_thinking : left_if;
  put_forks = left_if & (states[left] = HUNGRY &
    states[leftleft] != EATING & states[leftright] != EATING) :
    left_state_eating;
  put_forks = left_if & !(states[left] = HUNGRY &
    states[leftleft] != EATING & states[leftright] != EATING) :
    right_if;
  put_forks = left_state_eating : left_up_sem;
  put_forks = left_up_sem : right_if;
  put_forks = right_if & (states[right] = HUNGRY &
    states[rightleft] != EATING & states[rightright] != EATING) :
    right_state_eating;
  put_forks = right_if & !(states[right] = HUNGRY &
    states[rightleft] != EATING & states[rightright] != EATING) :
    up_mutex;
  put_forks = right_state_eating : right_up_sem;
  put_forks = right_up_sem : up_mutex;
  put_forks = up_mutex : end;
  put_forks = end : begin;
  1 : put_forks;
esac;
next (states[left]) := case
   put_forks = left_state_eating : EATING;
   1 : states[left];
   esac;
next (states[right]) := case
   put_forks = right_state_eating : EATING;
   1 : states[right];
   esac;
next (sems[left]) := case
   put_forks = left_up_sem : 1;
   1 : sems[left];
   esac;
next (sems[right]) := case
   put_forks = right_up_sem : 1;
   1 : sems[right];
   esac;
SMV : Model : Philosopher : test

init (insns) := thinking_;
next (insns) := case
  insns = thinking_ : take_forks_;
  insns = take_forks_ & take_forks = end : eating_;
  insns = eating_ : put_forks_;
  insns = put_forks_ & put_forks = end : thinking_;
  1 : insns;
esac;
-- specification AG (AF ph_0.eating & AF ph_1.eating & AF... is true
-- specification AG ((ph_0.eating -> !ph_4.eating & !ph_1... is true
-- specification AG (EF (ph_0.eating & ph_2.eating) & EF ... is true

resources used:
user time: 44.68 s, system time: 0.38 s
BDD nodes allocated: 268703
Bytes allocated: 5439488
BDD nodes representing transition relation: 45646 + 52
reachable states: 149494 (2^17.1897) out of 1.51447e+17 (2^57.0716)
SMV : Checking : 4

-- specification AG (AF ph_0.eating & AF ph_1.eating & AF... is true
-- specification AG ((ph_0.eating -> !ph_3.eating & !ph_1... is true
-- specification AG (EF (ph_0.eating & ph_2.eating) & EF ... is true

resources used:
user time: 2.98 s, system time: 0.01 s
BDD nodes allocated: 127874
Bytes allocated: 3211264
BDD nodes representing transition relation: 29363 + 42
reachable states: 16450 (2^14.0058) out of 6.37405e+13 (2^45.8573)
Java : Overview

• Implemented the pseudocode
  – 4 classes, 100 LOC
  – Used semaphore class from
    • http://www.dcs.napier.ac.uk/~shaun/rtse/labs/lab04.html

• Tested with 5, 10, and 100 philosophers for 10K cycles
  – When a philosopher was eating, its neighbors weren’t
    • Exclusive use of resources
/**
 * Shared class so don't have to pass arguments to Philosopher objects.
 */
class Shared {

    final static int THINKING = 0;
    final static int HUNGRY = 1;
    final static int EATING = 2;

    final static int NUM_PS = 5;

    final static Semaphore mutex = new Semaphore (1);
    final static Philosopher p[] = new Philosopher[NUM_PS];
    final static Semaphore[] sems = new Semaphore[NUM_PS];
    final static int[] state = new int[NUM_PS];

    final static int NUM_CYCLES = 10000; //TESTING
    final static boolean[] isEating = new boolean[NUM_PS]; //TEST
    final static Thread[] threads = new Thread[NUM_PS]; //TEST
}
public class DiningPhilosophers extends Shared {

    public static void main(String[] argv) {

        for (int i = 0; i < NUM_PS; i++) {
            sems[i] = new Semaphore (0);
            state[i] = THINKING;
            p[i] = new Philosopher (i);
        }

        for (int i = 0; i < NUM_PS; i++) (threads[i] = new Thread (p[i])).start();
    }
}
/**
 * Philosopher thread.
 */

class Philosopher extends Shared implements Runnable {

    private int id;

    Philosopher (int i) {
        id = i;
    }

    /**
     * Executed when thread is started.
     */
    public void run() {
        for (int i = 0; i < NUM_CYCLES; ++i) {
            think();
            take_forks();
            eat();
            put_forks();
        }
        System.out.println ("Philosopher " + id + " done.");  //TEST
    }

    ...
}
private void take_forks() {
    mutex.down();
    state[id] = HUNGRY;
    test(id);
    mutex.up();
    sems[id].down();
}

private void put_forks() {
    mutex.down();
    state[id] = THINKING;
    test(LEFT(id));
    test(RIGHT(id));
    mutex.up();
}
private void think() {} 

private void eat() {
    //TEST: Everything below is for testing only
    isEating[id] = true;
    threads[id].yield(); //yield so other threads try to enter, good test
    if (isEating[LEFT(id)] || isEating[RIGHT(id)]) { //exit if error
        System.out.println("Error, neighbors should not eat right now! " +
                       LEFT(id) + " " + id + " " + RIGHT(id));
        System.exit(0);
    }
    isEating[id] = false;
}

private void test (int i) {
    if (state[i]==HUNGRY && state[LEFT(i)]!=EATING &&
        state[RIGHT(i)]!=EATING) {
        state[i] = EATING;
        sems[i].up();
    }
}

private int LEFT (int i) {return (i + NUM_PS - 1) % NUM_PS;}

private int RIGHT (int i) {return (i + 1) % NUM_PS;}

/**
   * Semaphore class.
   * Taken from http://www.dcs.napier.ac.uk/~shaun/rtse/labs/lab04.html
   */
   
class Semaphore {

    private int count;

    Semaphore (int n) {
        this.count = n;
    }

    synchronized void down() {
        while(count == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
            count--;
        }
    }

    synchronized void up() {
        count++;
        notify(); //wakeup first thread that is blocking
    }
}
<table>
<thead>
<tr>
<th>Java : Test : 5</th>
<th>Java : Test : 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philosopher 0 done.</td>
<td>Philosopher 73 done.</td>
</tr>
<tr>
<td>Philosopher 2 done.</td>
<td>Philosopher 75 done.</td>
</tr>
<tr>
<td>Philosopher 4 done.</td>
<td>Philosopher 77 done.</td>
</tr>
<tr>
<td>Philosopher 1 done.</td>
<td>Philosopher 79 done.</td>
</tr>
<tr>
<td>Philosopher 3 done.</td>
<td>Philosopher 71 done.</td>
</tr>
<tr>
<td>Philosopher 0 done.</td>
<td>Philosopher 67 done.</td>
</tr>
<tr>
<td>Philosopher 1 done.</td>
<td>Philosopher 0 done.</td>
</tr>
<tr>
<td>Philosopher 3 done.</td>
<td>Philosopher 2 done.</td>
</tr>
<tr>
<td>Philosopher 5 done.</td>
<td>Philosopher 65 done.</td>
</tr>
<tr>
<td>Philosopher 7 done.</td>
<td>...</td>
</tr>
<tr>
<td>Philosopher 2 done.</td>
<td>Philosopher 80 done.</td>
</tr>
<tr>
<td>Philosopher 4 done.</td>
<td>Philosopher 82 done.</td>
</tr>
<tr>
<td>Philosopher 6 done.</td>
<td>Philosopher 84 done.</td>
</tr>
<tr>
<td>Philosopher 8 done.</td>
<td>Philosopher 86 done.</td>
</tr>
<tr>
<td>Philosopher 9 done.</td>
<td>Philosopher 88 done.</td>
</tr>
<tr>
<td>Philosopher 73 done.</td>
<td>Philosopher 90 done.</td>
</tr>
<tr>
<td>Philosopher 75 done.</td>
<td>Philosopher 92 done.</td>
</tr>
<tr>
<td>Philosopher 77 done.</td>
<td>Philosopher 94 done.</td>
</tr>
<tr>
<td>Philosopher 79 done.</td>
<td>Philosopher 96 done.</td>
</tr>
<tr>
<td>Philosopher 71 done.</td>
<td>Philosopher 98 done.</td>
</tr>
</tbody>
</table>