Testing

- Motivation
- JCStress
- Exhaustive Testing using Modelchecking
  - Spin
  - Promela
  - LTL
- Static Analysis
  - Findbugs

Demo Counter (Lecture 2 Recap)

```
public class Counter {
  private volatile int i = 0;
  public void inc() { i++; }
  public long getCount() { return i; }
}

class R implements Runnable {
  private Counter c;
  public R (Counter c) { this.c = c; }

  public void run() {
    for (int i = 0; i < 10; i++) {
      c.inc();
    }
  }
}
```

Demo Counter (Lecture 2 Recap)

```
class DemoCounter {
  public static void main(String[] args) {
    Counter c = new Counter();
    Runnable r = new R(c);
    Thread t0 = new Thread(r); Thread t1 = new Thread(r);
    t0.start(); t1.start();

    try {
      t0.join(); t1.join();
    } catch (InterruptedException e) {}

    System.out.println(c.getCount());
  }
}
```

What is the smallest possible value? Give it a deep thought!

Testing

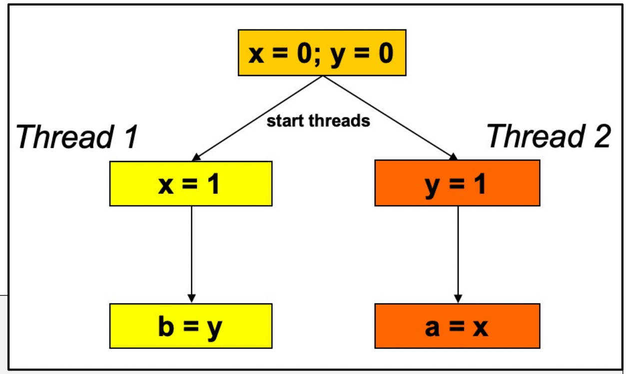
- **Motivation**
- **JCStress**
- **Exhaustive Testing using Modelchecking**
  - Spin
  - Promela
  - LTL
- **Static Analysis**
  - SpotBugs

Java Concurrency Stress tests - jcstress

- **Experimental harness and a suite of tests to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware**
- **Part of the OpenJDK testing infrastructure**
- **Works with user annotated classes and methods from which testrunners are generated (APT Annotation Processing Tool)**
- **Tests are invoked multiple times in order to provoke different outcomes**
  - Nondeterministic scheduling
  - Optimized, deoptimized invocations

http://openjdk.java.net/projects/code-tools/jcstress/

Example Test




```
@JCStressTest
@Description("Tests racy assignments.")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE,      desc = "t1 first")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE,      desc = "t2 first")
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE,      desc = "Interleaving")
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE_SPEC, desc = "JMM issue.")
@State
public class InterleavingTest {
    private int x = 0, y = 0;
    private int a = 0, b = 0;

    @Actor                                      @Actor
    public void thread1() {                     public void thread2() {
      x = 1;                                      y = 1;
      b = y;                                      a = x;
    }                                           }

    @Arbiter
    public void observe(II_Result res) {
        res.r1 = a;
        res.r2 = b;
    }
}
```

Worksheet jcstress: Exam Question

```java
public class JMM {
   private AtomicInteger ai = new AtomicInteger(5);
   private int i = 1;

   public void run() {
      new Thread(() -> {
         i++;
         ai.set(i);
      }, "T1").start();

      new Thread(() -> {
         int _i = i;          // (1)
         int _ai = ai.get(); // (2)
         System.out.println(_i + " " + _ai);
      }, "T2").start();
   }
   public static void main(String[] args) { new JMM().run(); }
}
```

Testing

- **Motivation**
- **JCStress**
- **Exhaustive Testing using Modelchecking**
  - Promela
  - Spin
  - LTL
- **Static Analysis**
  - SpotBugs

Computations as State Transition Systems

- **Computations can be interpreted as a sequence of steps from one program state to the next state**
- **Each program state consists of**
  - Program counter (PC) of each process
  - Value of each global variable
  - Value of each local variable
- **Parallel computations can be modeled by constructing a graph of all possible interleavings**

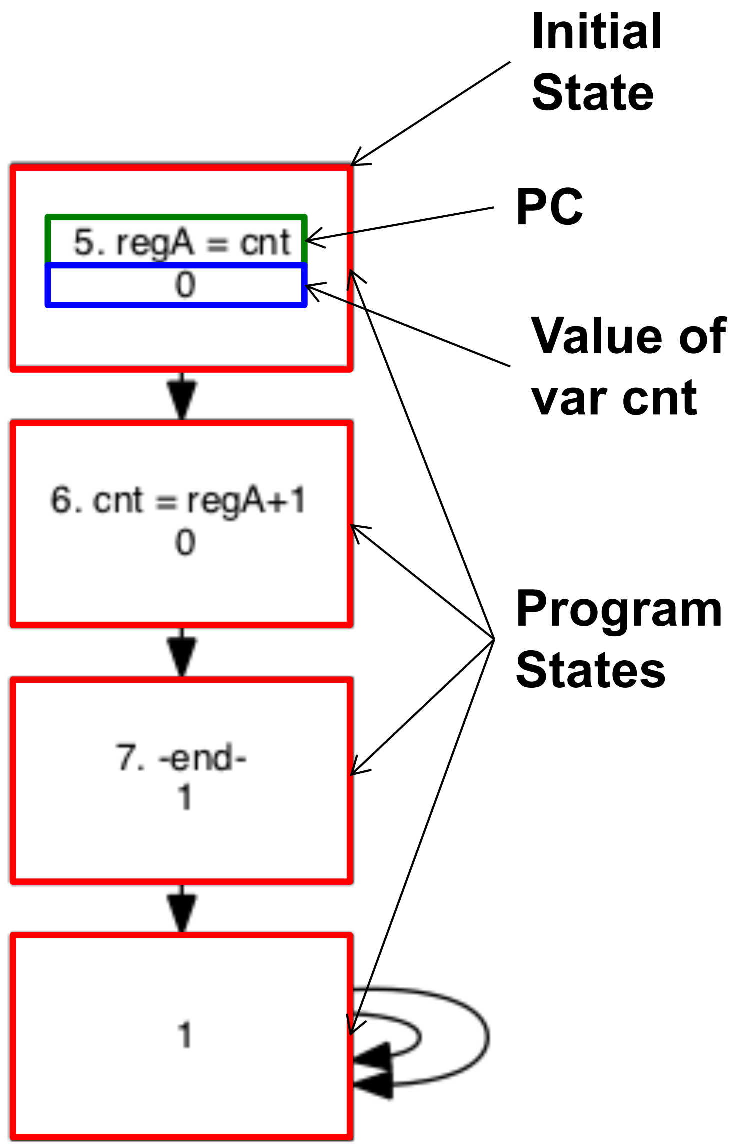
Interleaving

- **Single process**

```
1|   int cnt;
2|
3|   active proctype A() {
4|    int regA;
5|    regA = cnt;
6|    cnt = regA + 1
7|   }
```

every statement is atomic

PROMELA
specification language to model finite-state systems

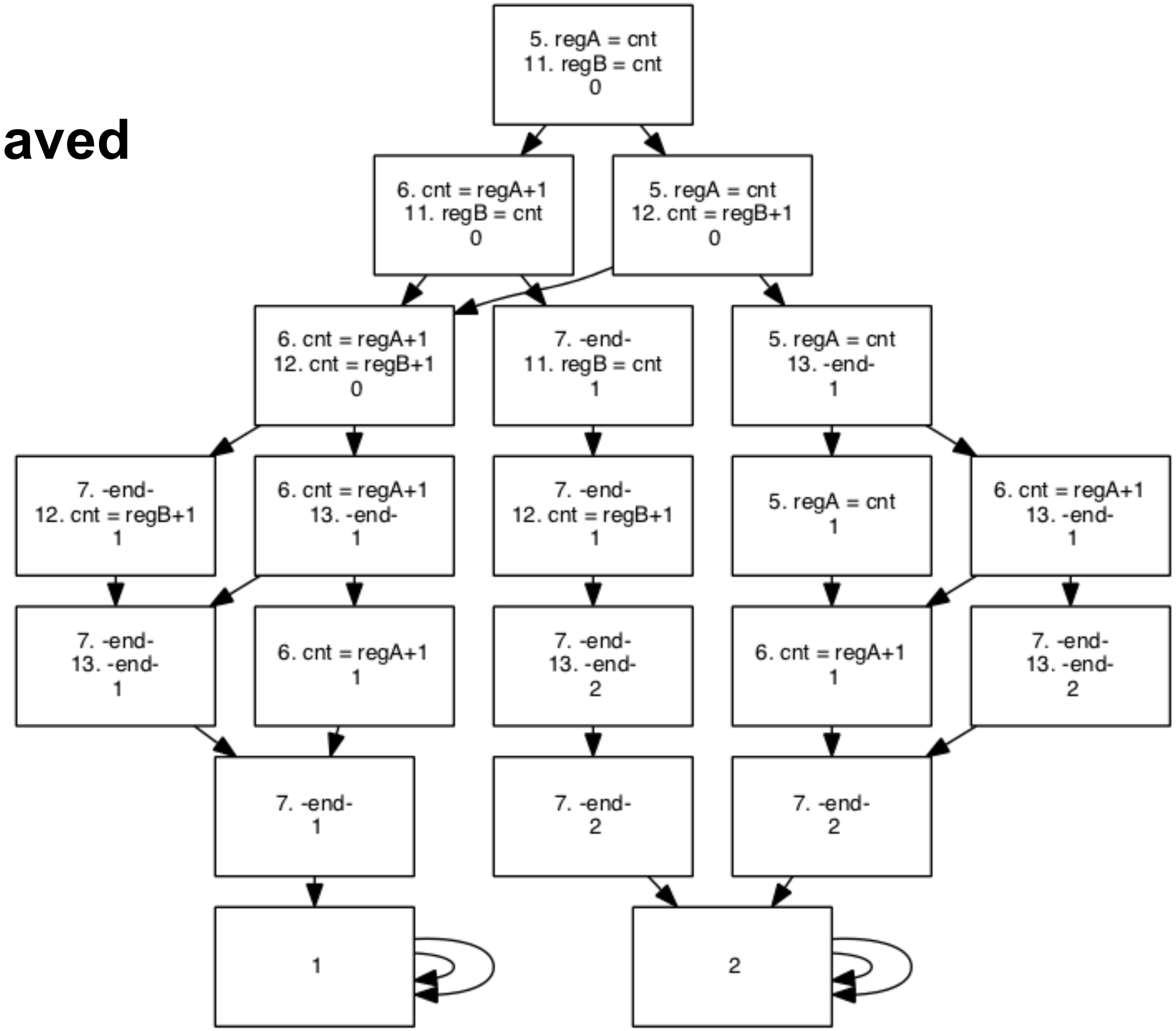


4 States

Interleaving

- **Two processes interleaved**

```
 1|  int cnt;
 2|
 3|  active proctype A() {
 4|   int regA;
 5|   regA = cnt;
 6|   cnt = regA + 1
 7|  }
 8|
 9|  active proctype B() {
10|    int regB;
11|    regB = cnt;
12|    cnt = regB + 1;
13|  }
```



22 States

Interleaving

- **Three processes interleaved**

```
int cnt;
```

```

active proctype A() {
 int regA;
 regA = cnt;
 cnt = regA + 1
}
```

```

active proctype B() {
  int regB;
  regB = cnt;
  cnt = regB + 1;
}
```

```

active proctype C() {
  int regC;
  regC = cnt;
  cnt = regC + 1;
}
```

159 States

Interleaving

- **Four processes interleaved**

```
int cnt;
```

```

active proctype A() {
  int regA;
  regA = cnt;
  cnt = regA + 1
}
```

```

active proctype B() {
  int regB;
  regB = cnt;
  cnt = regB + 1;
}
```

```

active proctype C() {
  int regC;
  regC = cnt;
  cnt = regC + 1;
}
```

```

active proctype D() {
  int regD;
  regD = cnt;
  cnt = regD + 1;
}
```

1465 States

Exhaustively Testing All Interleavings

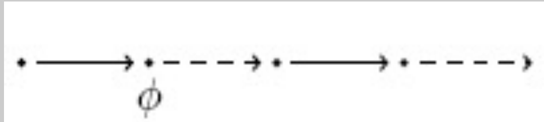
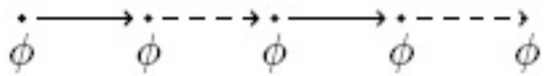
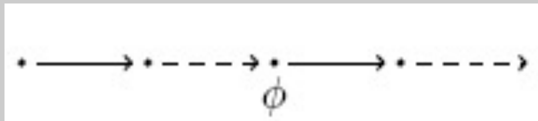
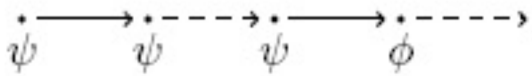
- **Idea: Check all possible outcomes**
- **Problem: Can't do with JVM**
- **Solution: Spin Modelchecker**

- **Approach**
  1. Model the program in PROMELA
  2. State some assertions
  3. Use SPIN to check all possible outcomes

```
Full statespace search for:
        never claim              + (minCntValue)
        assertion violations     + (if within scope of claim)
        acceptance   cycles      + (fairness disabled)
        invalid end states       - (disabled by never claim)

State-vector 52 byte, depth reached 169, errors: 1
```

```
#define N 10
byte cnt; // default 0

active [2] proctype Thread() {
  byte i;
  do
  :: (i < N) ->
     byte reg;
     reg = cnt;
     cnt = reg + 1;
     i = i + 1;
  :: else -> break;
  od
}

ltl minCntValue {
  <>([] (cnt >= 10))
}
```

Linear Temporal Logic - LTL

Textual	Symbolic	Explanation	Diagram
X ϕ	○ ϕ X ϕ	neXt: ϕ has to hold in the next state.	
G ϕ	□ ϕ [ ] ϕ	Globally: ϕ has to hold on the entire subsequent path.	
F ϕ	◇ ϕ <> ϕ	Finally: ϕ eventually has to hold (somewhere on the subsequent path).	
ψ U ϕ	ψ U ϕ ψ U ϕ	Until: ψ has to hold at least until ϕ, which holds at the current or a future position.	

Source: http://en.wikipedia.org/wiki/Linear_temporal_logic

LTL Properties

- **Safety properties**
  - **G ¬ϕ**
  - Something bad never happens
  - Example: Never more than one process in the critical section
    G (mutex <= 1)
- **Liveness**
  - **G (F ϕ)**
  - Something good keeps happening
  - Example: A process enters the critical section repeatedly
    G (F p[0]@critical)

Example: Mutual Exclusion (1)

FAILED

```
bool flag;  /* signal entering/leaving the section */
byte mutex; /* # procs in the critical section. */

proctype P(bit i) {
  flag == false ->
  flag = true;
  mutex++;
  mutex--;
  flag = false;
}

ltl mutexCheck {
 [] (mutex <= 1)
}

init {
  run P(0); run P(1);
}
```

Blocks until condition is true: while(! (flag == false)) skip;

The init process is active by default.

Example: Mutual Exclusion (2)

FAILED

```
bool aWant, bWant; /* signal entering/leaving the section */
byte mutex;        /* # of procs in the critical section. */
```

```
active proctype A() {
  aWant = true;
  bWant == false ->
  mutex++;
  mutex--;
  aWant = false;
}
```

```
active proctype B() {
  bWant = true;
  aWant == false ->
  mutex++;
  mutex--;
  bWant = false;
}
```

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

Concurrent process which checks that in every possible state (mutex != 2)

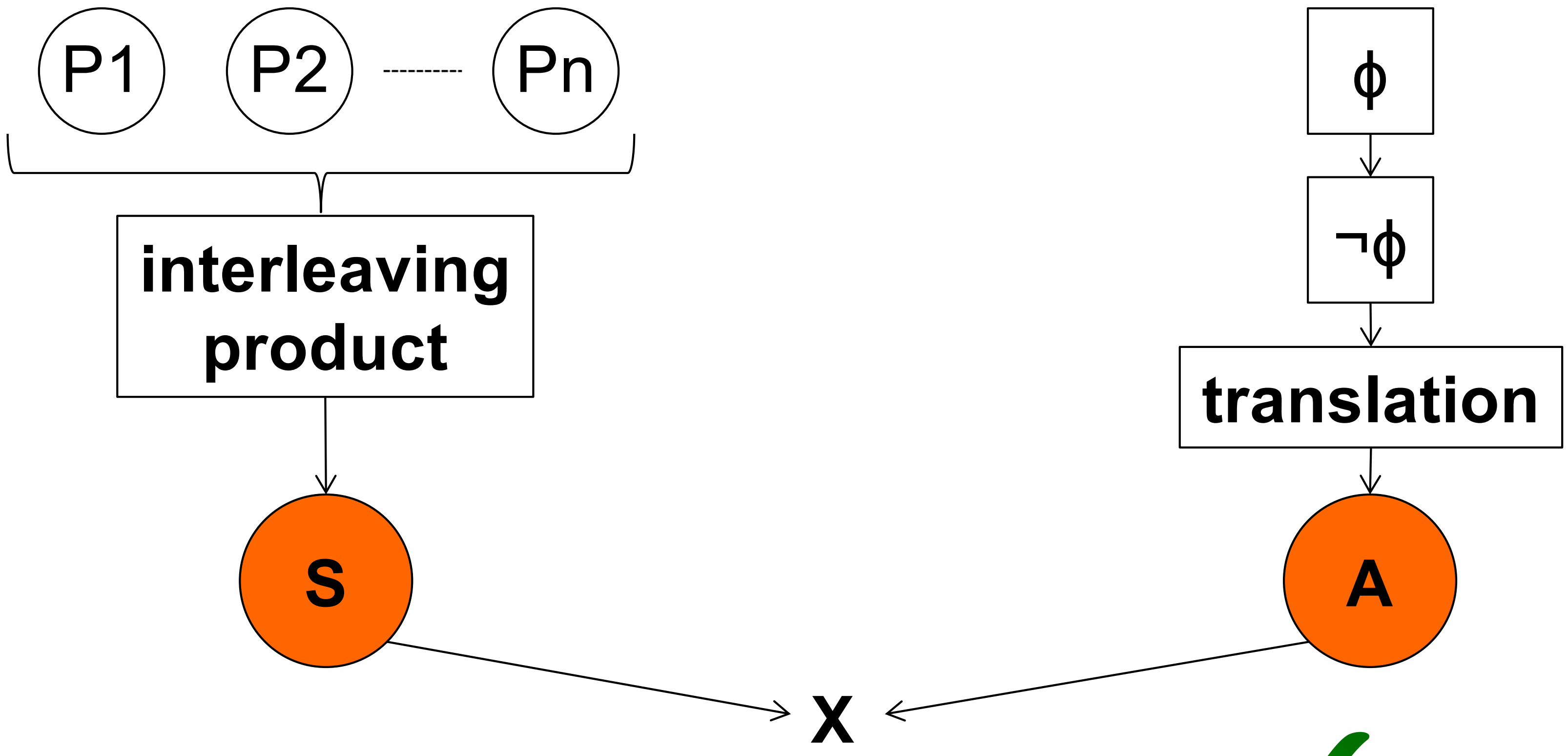
Example: Mutual Exclusion (3)

Peterson [1981] ✓

```
bool aWant, bWant; /* signal entering/leaving the section */
byte mutex;        /* # of procs in the critical section. */
pid turn;          /* who's turn is it? */

active proctype A() {
  assert(_pid == 0);
  aWant = true;
  turn = 1 - _pid;
  bWant==false || (turn==_pid) ->
  mutex++;
  mutex--;
  aWant = false;
}

active proctype B() {
  assert(_pid == 1);
  bWant = true;
  turn = 1 - _pid;
  aWant==false || (turn==_pid) ->
  mutex++;
  mutex--;
  bWant = false;
}

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

_pid returns the id of the executing process

How It Works (approx.)



If intersection X is empty ✓
otherwise: counter example

Spin Usage

Model (Counter.pml)

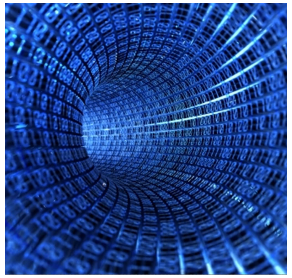
```
#define N 10
byte cnt; // default 0

active [2] proctype Thread() {
  byte i;
  do
  :: (i < N) ->
     byte reg;
     reg = cnt;
     cnt = reg + 1;
     i = i + 1;
  :: else -> break;
  od
}

ltl minCntValue {
  <>([] (cnt >= 10))
}
```

`spin -a Counter.pml` →

Verifier source (pan.c)

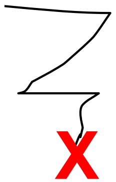
`cc -o pan pan.c` ↓

Verifier binary (pan)



← `./pan -a`

Counter example (Counter.pml.trail)



Testing

- **Motivation**
- **ConTest**
- **JCStress**
- **Exhaustive Testing using Modelchecking**
  - Promela
  - Spin
  - LTL
- **Static Analysis**
  - SpotBugs

SpotBugs

- **SpotBugs https://spotbugs.github.io/**
  - Applied in many large Java applications as part of the quality assurance
  - Complementary to unit tests

- **Static analysis tool**
  - Works by analyzing the static structure of a program
    not by executing code
  - Detectors for a wide range of common Java bug patterns
    Including concurrency bug patterns
    https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#multithreaded-correctness-mt-correctness

Summary: Testing

Testing concurrent software is difficult!

- **JCStress**
  - Repeatedly executing unit tests
- **SpotBugs**
  - Static program analysis
- **Spin**
  - Model checking (exhaustive testing of all possible executions)