i219 Software Design Methodology
11. Software model checking

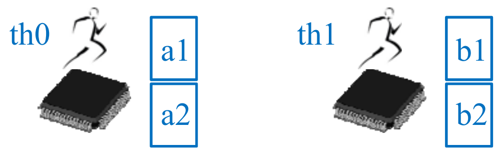
Kazuhiro Ogata (JAIST)

Outline of lecture

- Concurrency
- Model checking
- Java Pathfinder (JPF)
- Detecting race condition
- Bounded buffer problem
  - Detecting deadlock
  - Detecting assertion violation

Concurrency (1)

If a multithreaded program in which two threads (th0 and th1) may run in parallel is executed by a multi-core computer, the two threads may be truly running in parallel on two different cores.



For example, suppose that th0 performs two basic things a1 and a2 in this order and th1 performs two basic things b1 and b2 in this order. Then, while th0 is performing a1, th1 is performing b1 or b2 at the same time, and while th0 is performing a2, th1 is performing b1 or b2.

Concurrency (2)

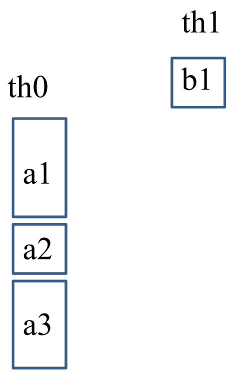
Since it is not easy to deal with true concurrency, concurrency is expressed as interleaving of basic things (called atomic actions or state transitions) performed by threads. For the two thread example on the previous page, since each thread performs two atomic actions (or state transitions) and each execution consists of four atomic actions, there are ${}_4C_2$ (= 6) possible execution sequences (called traces):

1. a1, a2, b1, b2
2. a1, b1, a2, b2
3. a1, b1, b2, a2
4. b1, a1, a2, b2
5. b1, a1, b2, a2
6. b1, b2, a1, a2

Concurrency (3)

Let us consider this simple multithreaded program:

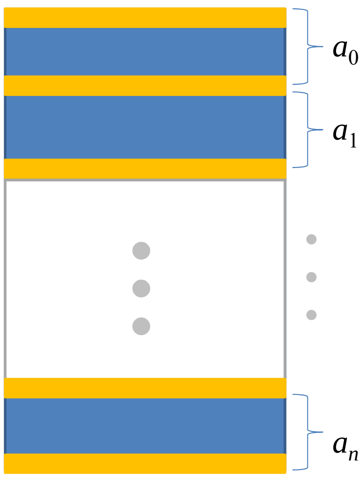
```
public class SimpInc extends Thread {
    private static int count = 0;
    private static int count2 = 0;
    public void run() { count2++; }
    public static void main(String[] args)
        throws InterruptedException {
        Thread t = new SimpInc();
        t.start();
        count++;
        t.join();
        System.out.println("count: " + count);
        assert count == 2;
    }
}
```



b1 can be performed after a1 and before a3 and then there are two possible traces.

Concurrency (4)

Given a multithreaded program:



where

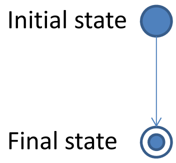
is a point (such as a message passing expression) that has something to do with thread scheduling, such as t.start() and t.join(), or where objects (or values) shared by multiple threads are accessed non-mutually exclusively, and $a_0, a_1, \ldots, a_n$ are fragments of the program that do not have such a point anywhere except for the beginning and end.

Then, $a_0, a_1, \ldots, a_n$ are treated atomic actions, some of which may be executed multiple times by one thread or multiple ones.

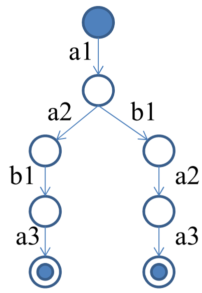
Concurrency (5)

From a multithreaded program, what is called a computation tree is made, in which nodes are program states and edges are executions of atomic actions (called transitions). The root is the very initial state of the program.

An ordinal program in which there is one thread:

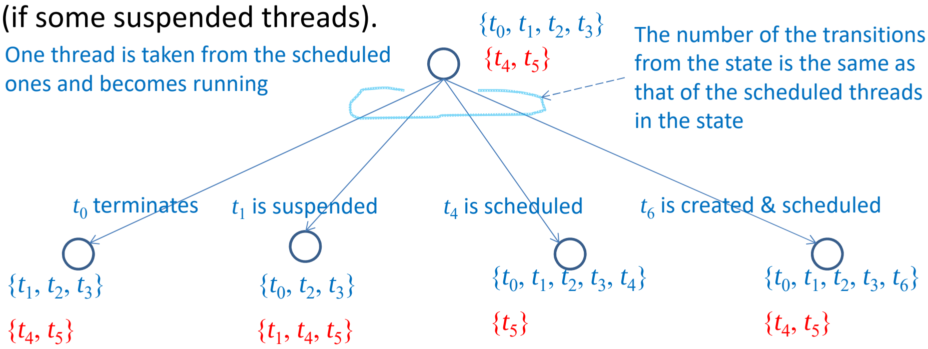


SimpInc:



Concurrency (6)

In each state of a computation tree, there is 0 or more threads scheduled and 0 or more threads suspended. In the initial state, there is only one scheduled thread (the main thread executing the static main(…) method). If there are no scheduled threads, the state is either a final state (if no suspended threads) or what is called a deadlock state (if some suspended threads).

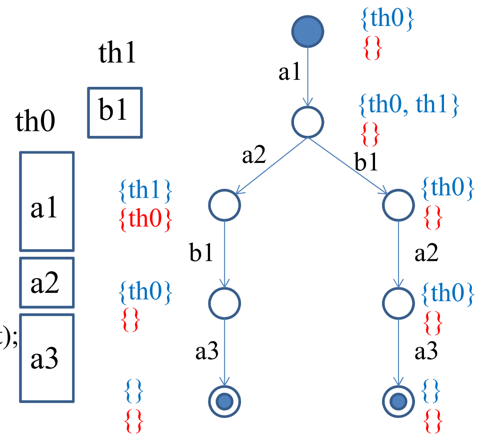


Note that a transition may not alter the scheduled and suspended threads at all.

Concurrency (7)

Let us revisit this simple multithreaded program:

```
public class SimpInc extends Thread {
    private static int count = 0;
    private static int count2 = 0;
    public void run() { count2++; }
    public static void main(String[] args)
        throws InterruptedException {
        Thread t = new SimpInc();
        t.start();
        count++;
        t.join();
        System.out.println("count: " + count);
        assert count == 2;
    }
}
```



Model checking

Exhaustively traverses all possible traces of a given multithreaded program to find a state or a path to a state in which some desired properties, such as deadlock freedom, are not fulfilled.

Testing may not traverse all possible traces of a given multithreaded program because the scheduler of the Java virtual machine cannot be controlled by ordinary programs.

Therefore, model checking makes it possible to detect some flaws that can never be detected by testing.

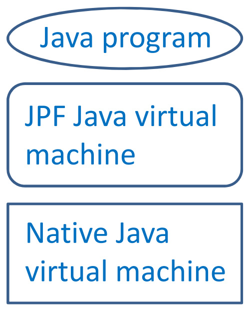
Java Pathfinder (JPF) (1)

A model checker for Java programs. Also equipped with some other functionalities. Used for model checking multithreaded Java programs in this course.

JPF does not control the scheduler of the native Java virtual machine but has its own virtual machine that is an ordinal Java program running on the native Java virtual machine.

Java programs model checked by JPF run on the JPF virtual machine.

JPF controls the scheduler of its own virtual machine to take into account all possible traces of a given multithreaded program.



Java Pathfinder (JPF) (2)

Let us model check SimpInc with JPF. JPF checks if each assertion holds in each possible trace. If JPF finds a state in which some assertion does not hold, it shows a trace leading to the state.

To this end, we prepare a .jpf file (SimpInc.jpf in this case) whose contents are as follows:

program to be checked

the current directory (folder) is added to both classpath & sourcepath

asking JPF to display error, trace & snapshot information when JPF finds something wrong

```
target = SimpInc
classpath+=.
sourcepath+=.
report.console.property_violation=error,trace,snapshot
```

Then, we can model check the program with JPF as follows:

```
% javac SimpInc.java
% jpf SimpInc.jpf
```

Java Pathfinder (JPF) (3)

The assertion does not hold in any trace, and then JPF finds a state in which it does not hold and displays a trace leading to the state:

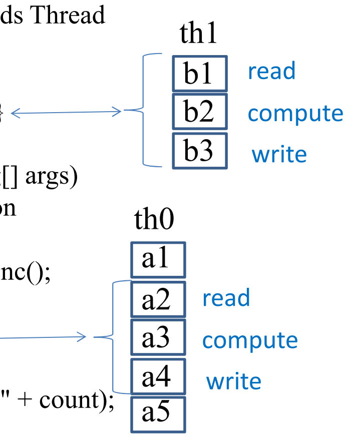
```
=============================================================== trace #1
---------------------------------------------------- transition #0 thread: 0     a1
… Source code fragment corresponding to
---------------------------------------------------- transition #1 thread: 0
… Source code fragment corresponding to                                          a2
---------------------------------------------------- transition #2 thread: 1
… Source code fragment corresponding to                                          b1
---------------------------------------------------- transition #3 thread: 0
… Source code fragment corresponding to                                          a3
=============================================================== snapshot #1
… Some info shown here about the state in which
  something wrong happened
=============================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError  at
SimpInc.main(SimpInc...."
```

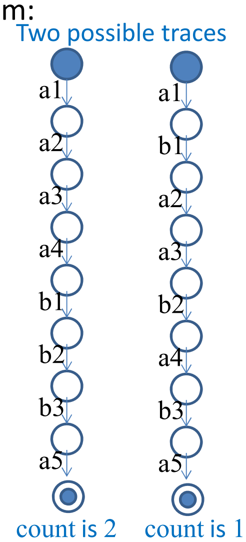

Java Pathfinder (JPF) (4)

Let us consider another simple multithreaded program:

```
public class SimpConcInc extends Thread
{
   private static int count = 0;
   public void run() { count++; }

   public static void main(String[] args)
      throws InterruptedException
   {
      Thread t = new SimpConcInc();
      t.start();
      count++;
      t.join();
      System.out.println("count: " + count);
      assert count == 2;
   }
}
```





Java Pathfinder (JPF) (5)

JPF finds a state in which the assertion does not hold and shows a trace leading to the state:

```
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0   a1
…
------------------------------------------------------ transition #1 thread: 1   b1   read count (0)
…
------------------------------------------------------ transition #2 thread: 0   a2   read count (0)
…
------------------------------------------------------ transition #3 thread: 0   a3   increment (1)
…
------------------------------------------------------ transition #4 thread: 1   b2   increment (1)
…
------------------------------------------------------ transition #5 thread: 0   a4   write count (1)
…
------------------------------------------------------ transition #6 thread: 1   b3   write count (1)
…
------------------------------------------------------ transition #7 thread: 0   a5
…
====================================================== snapshot #1
…                                                    count is 1
====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError  at
SimpConcInc.main(Simp..."
```


Detecting race condition (1)

```
public class FCounter {
    private static int x = 0;
    public static int get() { return x; }
    public synchronized void inc() { x++; } }

public class UnsafeInc extends Thread {
    public void run() { (new FCounter()).inc(); }
    public static void main(String[] args)
        throws InterruptedException {
        Thread t1 = new UnsafeInc();
        Thread t2 = new UnsafeInc();
        t1.start();  Thread.sleep(1000);  t2.start();
        t1.join();  t2.join();
        System.out.println("count: " + FCounter.get());
        assert FCounter.get() == 2; } }
```

- ✓ One static field x is shared by all objects of FCounter.
- ✓ Since the two threads t1 & t2 use objects of FCounter, t1 & t2 also share the static field x in FCounter.
- ✓ Since inc() is synchronized in which x is incremented, there seems no race condition in the program.
- ✓ No matter how many times it is launched (tested), what is displayed is 2.

✓ But, … you see?

Detecting race condition (2)

```
------------------------------------------------------ transition #6 thread: 2
...
FCounter.java:4            : public synchronized void inc() { x++; }    read
------------------------------------------------------ transition #7 thread: 1
...
FCounter.java:4            : public synchronized void inc() { x++; }    read
------------------------------------------------------ transition #8 thread: 1
...
FCounter.java:4            : public synchronized void inc() { x++; }    compute
------------------------------------------------------ transition #9 thread: 2
...
FCounter.java:4            : public synchronized void inc() { x++; }    compute
------------------------------------------------------ transition #10 thread: 1
...
FCounter.java:4            : public synchronized void inc() { x++; }    write
...
------------------------------------------------------ transition #11 thread: 0
...
------------------------------------------------------ transition #12 thread: 2
...
FCounter.java:4            : public synchronized void inc() { x++; }    write
...
=========================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError  at
UnsafeInc.main(Unsafe..."
```

Why?

Detecting race condition (3)

Let us take a close look at the program:

```
public class FCounter {
   private static int x = 0; … public synchronized void inc() { x++; } }

public class UnsafeInc extends Thread {
   public void run() { (new FCounter()).inc(); }
   public static void main(String[] args) … { …
      t1.start();  …  t2.start(); … }
```

Each thread creates a new object of FCounter and sends inc() to the object.

Let c_i be the object of FCounter created by ti (i = 1,2).

When t1 sends inc() to c_1, t1 successfully acquires the lock associated with c_1 because there is no thread that has the lock.

When t2 sends inc() to c_2, t2 successfully acquires the lock associated with c_2 because c_2 is different from c_1 and then there is no thread that has the lock.

Therefore, t1 and t2 may increment x simultaneously.

Detecting race condition (4)

A possible remedy

```
public class GCounter {
   private static int x = 0;
   private static Object lock = new Object();
   public static int get() { return x; }
   public void inc() { synchronized (lock) { x++; } } }

public class SafeInc extends Thread {
   public void run() { (new GCounter()).inc(); }
   public static void main(String[] args)  throws InterruptedException {
      Thread t1 = new SafeInc();  Thread t2 = new SafeInc();
      t1.start();  Thread.sleep(1000);  t2.start();
      t1.join();  t2.join();
      System.out.println("count: " + GCounter.get());
      assert GCounter.get() == 2;  }  }
```


Detecting race condition (5)

Another possible remedy

```
public class Counter {
   private int x = 0;
   public synchronized int get() { return x; }
   public synchronized void inc() { x++; }  }

public class SafeInc2 extends Thread {
   private static Counter counter = new Counter();
   public void run() { counter.inc(); }
   public static void main(String[] args)  throws InterruptedException {
      Thread t1 = new SafeInc2(); Thread t2 = new SafeInc2();
      t1.start();  Thread.sleep(1000);  t2.start();
      t1.join();  t2.join();
      System.out.println("count: " + counter.get());
      assert counter.get() == 2;  }  }
```

Bounded buffer problem (1)

Queue<E>, EmpQueue<E>, NeQueue<E>, MonitorBBuf<E>, Sender<E>, and Receiver<E> are the same as those used in lecture note 10. BBProb is as follows:

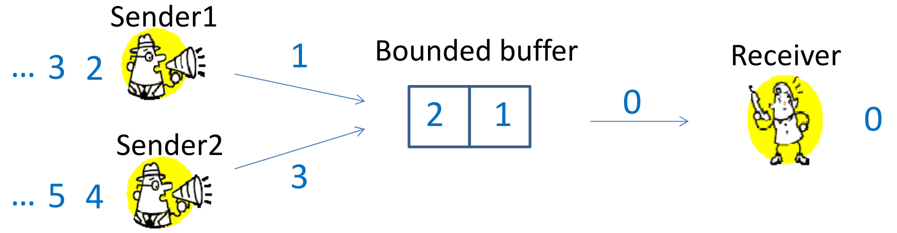
```
public class BBProb {
  public static void main(String[] args) throws InterruptedException {
    MonitorBBuf<Integer> buf = new MonitorBBuf<Integer>(2);
    …
    for (int i=0; i < 2; i++) msgsSent.add(i);
    …
    assert msgsReceived.equals(msgsSent); } }
```

The remaining parts … are the same as those used in lecture note 10.

JPF does not detect any flaws for this version.

Bounded buffer problem (2)

Let us revise the problem such that there are two senders.



MonitorBBuf<E> is renamed MonitorBBuf2<E> and modified as follows:

```
public class MonitorBBuf2<E> { …
  private List<E> log; …
  public MonitorBBuf2(int cap,List<E> log) { … this.log = log; }
  public synchronized void put(E e) … { …
    if (noe < capacity) { … log.add(e); … } }
  … }
```

Bounded buffer problem (3)

public class Sender2<E> extends Thread { … }
public class Receiver2<E> extends Thread { … }

MonitorBBuf2<E> is used in these classes.

BBProb is renamed BBProb2 and modified as follows:

```
public class BBProb2 { public static void main(String[] args) … {
    List<Integer> log = new ArrayList<Integer>();
    MonitorBBuf2<Integer> buf = new MonitorBBuf2<Integer>(2,log); …
    int nom = msgsSent.size()+msgsSent.size(); …
    Sender2<Integer> sender2 = new Sender2<Integer>(buf,msgsSent);
    … sender2.start(); … sender2.join(); …
    assert msgsReceived.equals(log);} }
```

MonitorBBuf2<E>, Sender2<E> & Receiver2<E> are used in the class.

JPF does not detect any flaws for this version as well.

Bounded buffer problem (4)

MonitorBBuf2<E> is renamed FMonitorBBuf1<E> and modified as follows:

```
public class FMonitorBBuf1<E> { …
  public synchronized void put(E e) … {
    if (noe >= capacity) this.wait(); … }
  … }
```

public class FSender1<E> extends Thread { … }
public class FReceiver1<E> extends Thread { … }

FMonitorBBuf1<E> is used in these classes.

public class FBBProb1 { … }

FMonitorBBuf1<E>, FSender1<E> & FReceiver1<E> are used in the class.

JPF detects a possible deadlock that may happen in this version.

Bounded buffer problem (5)

```
------------------------------------------------------ transition #72 thread: 1
…
FMonitorBBuf1.java:16         : this.wait();
…
```

sender

```
------------------------------------------------------ transition #82 thread: 2
…
FMonitorBBuf1.java:16         : this.wait();
…
```

sender2

```
------------------------------------------------------ transition #83 thread: 3
…
FMonitorBBuf1.java:31         : this.notifyAll();
…
```

receiver

```
------------------------------------------------------ transition #85 thread: 1
…
FMonitorBBuf1.java:17         : if (noe < capacity) {
FMonitorBBuf1.java:18         : queue = queue.enq(e);
…
```

puts 1 into the buffer

```
------------------------------------------------------ transition #110 thread: 2
…
FMonitorBBuf1.java:17         : if (noe < capacity) {
FMonitorBBuf1.java:23         : }
…
```

fails to put 1 into the buffer

Both senders terminate.

```
------------------------------------------------------ transition #114 thread: 3
…
FMonitorBBuf1.java:26         : this.wait();
…
```

waits forever here
deadlock

Bounded buffer problem (6)

```
public class FMonitorBBuf1<E> { …
 public synchronized void put(E e) … {
  if (noe >= capacity) this.wait(); … } … }
```

Suppose that Sender1 & Sender2 waits on this.wait(), and Receiver gets one element from the buffer and executes this.notifyAll(), waking up both Sender1 & Sender2.

After Receiver releases the lock l associated with the buffer, both Sender1 & Sender2 try to acquire l. Suppose that Sender1 acquires l and then Sender2 waits until l is released.

After l is released, Sender2 acquires l. Although now $<$ capacity does not hold at this moment, however, Sender2 proceeds, neither putting an element, such as 1, into the buffer nor executing this.notifyAll().

After that, Receiver waits on this.wait(), but this.notifyAll() will never be executed, namely that deadlock has been encountered.

Bounded buffer problem (7)

MonitorBBuf2<E> is renamed FMonitorBBuf2<E> and modified as follows:

```
public class FMonitorBBuf2<E> { …
  public synchronized E get() … {
    while (noe < 0) this.wait(); … } }
```

Receiver2<E> is renamed FReceiver2<E> and modified as follows:

```
public class FReceiver2<E> { …
  public void run() { …
    try { msgs.add(buf.get()); Thread.sleep(100); } … } }
```

FMonitorBBuf2<E> is used in this class and the following class:

```
public class FSender2<E> extends Thread { … }
```

FMonitorBBuf2<E>, FSender2<E> & FReceiver2<E> are used in the class.

```
public class FBBProb2 { … }
```


Bounded buffer problem (8)

JPF detects a possible trace in which the assertion does not hold.

```
…
msgsSent: [0, 1, 0, 1]
msgsReceived: [0, 1, 0, null]
Failure!
…
---------------------------------------------------- transition #130 thread: 3
…
 FMonitorBBuf2.java:24        : while (noe < 0)
 FMonitorBBuf2.java:26        : if (noe > 0) {
 FMonitorBBuf2.java:33        : return null;
```

receiver gets null from the buffer

```
while (noe < 0) this.wait();
```

is used instead of

```
while (noe <= 0) this.wait();
```

This is why there exists such a trace in which the assertion does not hold.

Summary

- Concurrency
- Model checking
- Java Pathfinder (JPF)
- Detecting race condition
- Bounded buffer problem
  - Detecting deadlock
  - Detecting assertion violation