Testing and Debugging Concurrent Software

Shmuel Ur
Why is Concurrent Testing Hard?

- **Concurrency introduces non-determinism**
  - Multiple executions of the same test may have different interleavings (and different results!)
  - an interleaving is the relative execution order of the program threads
  - Very hard to reproduce and debug
- No useful coverage measures for the interleaving space
- Typically appear only in specific configurations
  - Therefore commonly found by users
  - Require large configurations to test
- Represent only ~10% of the bugs but an un-proportional number are found late or by the customer -> Very expensive

The costly effort of testing concurrency at system level is seemingly unavoidable
The Classic Java Example – A Counter

Let’s consider the function increase(), which is a part of a class that acts as a counter

```java
public void increase()
{
    counter++;  
}
```

Although written as a single “increase” operation, the “++” operator is actually mapped into three JVM instructions `[load operand, increment, write-back]`
Counter Example – Continued

Thread A

... load operand
... increment
... write-back
... Context Switch

Thread B

... load operand
... increment
... write-back
... 

\[ \text{counter} = 8 \]

“Race conditions represent one of your biggest enemies when it comes to programming concurrent applications”

High Performance Java Platform Computing, Christopher & Thiruvathukal
Sun Java Series, 2000
Riddle I: How many bugs do you see?

```java
public class Helloworld{
    public static void main(String[] argv){
        Hello helloThread = new Hello();
        World worldThread = new World();
        helloThread.start();
        worldThread.start();

        try{Thread.sleep(1000);} catch(Exception exc){};
        System.out.print("
        ");
    }
}

class Hello extends Thread{
    public void run(){
        System.out.print("hello ");
    }
}

class World extends Thread{
    public void run(){
        System.out.print("\n");
        System.out.print("world");
    }
}
```
Riddle II: Understanding Synchronization Primitives

- Locks protect the code segment and not the shared data
  - Obtaining a lock when accessing the shared resources
- On an error path (e.g., an exception) does the system release the lock?
- Consider the following class: `class Conflict {
  Conflict(...) { synchronized(Conflict.class){...}; }
  void h(...) { synchronized(this){...}; }
  synchronized void g(...) {...; }
  void r(...) {...; }
  synchronized static void f(...) {...; }
} Can f || g, f || h, f || r, g || h, g || r, h || r cause a conflict?
How Much is \(1+10+100+1000+10000?\)

```java
static final int NUM = 5;
static int Global = 0;

public static void main(String[] argv){
    Global = 0;
    threads[0] = new Adder(1);
    threads[1] = new Adder(10);
    threads[2] = new Adder(100);
    threads[3] = new Adder(1000);
    threads[4] = new Adder(10000);

    // Start Threads
    for(int i=0; i<NUM; i++) { threads[i].start(); }

    try{ Thread.sleep(1000); } catch(Exception exc){}

    // Print Results
    System.out.println(Global);
}
```
class Adder

class Adder extends Thread

    int Local;

    Adder(int i){
        Local = i;
    }

    public void add(){
        example.Global += Local;
    }

    public void run(){
        add();
    }
Coverage Results without ConTest

Here are the legal tasks:

<table>
<thead>
<tr>
<th>ThreadA</th>
<th>ThreadB</th>
<th>ThreadC</th>
<th>ThreadD</th>
<th>ThreadE</th>
<th>TOTAL COVERAGE</th>
<th>UNIQUE COVERAGE</th>
<th>OUT OF</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Number of Tasks: 32
Covered: 1
Coverage Percentage: 3.125

Click on a column header to sort the rows by this column.

Print  Export  Close
Coverage Results with ConTest

<table>
<thead>
<tr>
<th>ThreadA</th>
<th>ThreadB</th>
<th>ThreadC</th>
<th>ThreadD</th>
<th>ThreadE</th>
<th>TOTAL COVERAGE</th>
<th>UNIQUE COVERAGE</th>
<th>OUT OF</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>47</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>68</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>34</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>19</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>62</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>66</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>19</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>57</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>46</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>60</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>107</td>
<td>1</td>
<td>1</td>
<td>100.0</td>
</tr>
</tbody>
</table>
method Add

Java Source code

    public void add(){
        example.Global += Local;
    }

Byte Code

Method void add()
    0 getstatic #3 <Field int Global>
    3 aload_0
    4 getfield #2 <Field int Local>
    7 iadd
    8 putstatic #3 <Field int Global>
   11 return
A Bug Published by the ConTest Project Team

A race condition is a possible source for a defect, since the value of the variable at the time of reading depends on the scheduling. However, not all race conditions are defects. For example, the following code swaps two integers. There is a race condition, but no defect, as the swapping occurs regardless of the interleaving.

class Change{
    static int x = 4, y = 5;
    //Used to implement a busy wait.
    static int z1 = -1, z2 = -1;
    //Swap the value of x and y concurrently
    public static void main(String args[ ]){
        (new Thread(new ChangeA( ))).start( );
        (new Thread(new ChangeB( ))).start( );
    }
}
class ChangeA implements Runnable{
    public void run( ){
        Change.z1 = Change.x;
        while(Change.z2 == -1)
            System.out.println("A is waiting");
        Change.x = Change.z2;}
}
class ChangeB implements Runnable{
    public void run( ){
        Change.z2 = Change.y;
        while(Change.z1 == -1)
            System.out.println("B is waiting");
        Change.y = Change.z1;}
}

It should be noted that race conditions are execution-dependent: a program might be in a race condition in one execution and not in another. Therefore, tools that detect races at run time (or by analyzing the trace of a given run) are likely to miss some potential data races.
Atomicity is Never Ensured

```java
static void transfer(Transfer t) {
    balances[t.fundFrom] -= t.amount;
    balances[t.fundTo] += t.amount;
}
```

**Expected Behavior:**
Money should pass from one account to another

**Observed Behavior:**
Sometimes the amount taken is not equal to the amount received

**Possible bug:**
Thread switch in the middle of money transfers
Atomicity is Never Ensured II

- Assume Atomic transfer (can't be implemented locally in Java)
  - balances[t.fundFrom] -= t.amount;
  - balances[t.fundTo] += t.amount;

- Assume a counting loop that checks if total remains the same

- Does it work now?
Scenario 1

The train enters the tunnel.
The semaphore automatically turns the light to red.
Signalman A sends a message to signalman B, that a train is in the tunnel.
The train exits the tunnel.
Scenario 2

The first train enters the tunnel.
The semaphore automatically turns the light to red.
Signalman A sends a message to signalman B, that a train is in the tunnel.
The second train approaches the tunnel and stops at the red light.
The first train exits the tunnel.
B sends a message to A, that the first train has exited the tunnel.
A sets the light to green.
The second train enters the tunnel.
Scenario 3

The first train enters the tunnel – the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

The second train approaches the tunnel.

Signalman A runs to the track and signals to the train to stop and sets the light to red, manually.

The first train exits the tunnel.

B sends a message to A, that the first train has exited the tunnel.

A sets the signal to green and lets the second train pass.

The second train enters and leaves the tunnel.
The first train enters the tunnel – the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

Signalman A runs to the track to get the light to red and signals other trains to stop.

But a second train arrives and enters the tunnel before he has time to change the light to red. The driver sees something but doesn’t have time to stop.

Signalman A sends another message to signalman B, that a train is in the tunnel.

The second train driver decides to back out of the tunnel.

The second train enters the tunnel.

The third train enters the tunnel.

The third train approaches the tunnel and stops.

The second and third trains collide.

The driver of the second train is suspicious that something is wrong and stops in the middle of the tunnel.
Bug Found by ConTest in Websphere Site Analyzer

- Crawler, a ~1000 line Java component, is part of the Websphere Site Analyzer used to perform content analysis of web sites

- Bug description:
  - if (connection != null) connection.setStopFlag();
    - Connection is checked to be !null
    - CPU is lost
    - Connection is set to null before CPU is regained
  - If this happens before connection.setStopFlag(); is executed, an exception is taken

- This bug was found while we were still testing ConTest
- This bug should (also) have been found in unit testing...
Some Typical Concurrent Bug Patterns
Not-Atomic

- An operation is assumed to be atomic but is actually not
  - Source code operations often seem to the inexperienced programmer to be atomic when they are not
  - Example: x++
Two-Stage-Access

Two stage access:

- We are given two tables
- To change a record in the second table, the first table is queried and then the second
- Each table is protected by a separate lock

```
lock [ First query  key1 -> key2 ]

window -> the tables can be changed here

lock [ Second query  key2 -> record to be changed ]
```
Wrong/No-Lock

Wrong lock or no lock
- Protection of thread one does not apply to thread two
- There is an access protocol that is not followed due to:
  - A new team member
  - An attempt to improve performance

Thread 1
Synchronized (o){
  x++;
}

Thread 2
x++;
Initialization-Sleep

- One example is adding `sleep()` statements to ensure that only the correct interleavings occur.
- Partial, non-consistent results are used by the thread that assumes that initialization is done.
Lost-Notify

- Losing notify: the notify is “lost” because it occurs before the thread executes the wait() primitive
- The gap was created because the programmer didn’t think the notify would occur before the wait

Thread 1

```java
synchronized (o){
    o.notifyAll();
    o.wait();
}
```

Thread 2

```java
synchronized (o){
    o.notifyAll();
}
```
Blocking-Critical-Section

- Blocking critical section
  - In the design of a critical section protocol we assume that the thread executing the critical section will eventually exit.
  - This assumption might be broken if the code is written by a third party or a different group.
Orphaned-Thread

The tale of the orphaned thread
- A single master thread drives actions of other threads
- Messages are put on the queue by the master thread and processed by the worker’s threads
- Abnormal termination of the master thread results in the remaining threads being orphaned
  - The system often blocks
Unintentional-Different-Thread

- A call to an API (typically a GUI API) is assumed to be in the same thread but is actually in a different thread causing the order of locking to sometimes change resulting in a deadlock
  - Nasty when you have a deadlock in a single thread program…
Condition-For-Wait

- Missing condition enclosing the wait
  - When returning from a wait the programmer forgets to check, or checks incorrectly if the reason for which he waited still holds
  - In Java 1.5 a wait can decide to terminate. Is your code ready?

- From http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#wait(long)
  - A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops, like this one:

```java
synchronized (obj) {
  while (<condition does not hold>)
    obj.wait(timeout); ... // Perform action appropriate to condition
}
```
Delaying messages with ConTest/UDP

Mode = Delay Direction = Out

Network

B

A

Java
UDP
IP
Link

ConTest/UDP

Java
UDP
IP
Link

ConTest/UDP
Delaying messages with ConTest/UDP

Mode = No-Noise

Application

ConTest/UDP

Java

UDP

IP

Link

Network

A

B

1 2 3 4

ConTest/UDP

Java

UDP

IP

Link

ConTest/UDP

Java

UDP

IP

Link

3 2 1
Losing messages with ConTest/UDP

B

ConTest/UDP

Java
UDP
IP
Link

Application

Mode = Block
Direction = Out

Network

A

ConTest/UDP

Java
UDP
IP
Link
Losing messages with ConTest/UDP
Some Interesting Observation

- In practice thread switches are few and far between
  - The probability that the previous bug will be found is low
  - Synchronization usually operate as a no-op
    - Removing all synchronizations usually will not impact testing results!
    - Not knowing the synchronization primitives exact definition does not impact testing but the program is incorrect
      - Exception in synchronization: do you still have the lock?
      - What is the synchronization on?

- Thread scheduling for small applications is almost deterministic in simple environment
  - Each environment has its own interleaving
    - Customer on the first day find bugs in well tested applications!
How Does a Noise Maker Work?

- **Find places which are “concurrently interesting”**
  - Places whose relative interleaving may change the result of the program: access to shared variables, synch primitives
  - Possibly use bug patterns to identify such places
    - Lost notify
    - Sleep()
    - Condition before synchronization

- **Modify the program by adding interleaving changing mechanisms**
  - Usually sleep(), yield() but more advance mechanisms exist
    - Make sure no new bug is introduced
  - Make sure that the interleaving change is random

- **Download** [http://www.alphaworks.ibm.com/tech/contest](http://www.alphaworks.ibm.com/tech/contest) to see one
With ConTest each Test Goes a Long Way
ConTest Debug Options – many other options will talk about some of them later

- Deadlock
  - Location of each thread
  - Cycle of waiting on locks
  - Lock discipline violation
  - Lock taking trace

- Lock Discipline – causing and healing
- **orange_box** – remember \{last_values_size\} of values and locations for each variable. Created for null pointer exception

- Replay – record and reuse replay information
- Talk with (socket/keyboard) ConTest while the application is running
Unit Testing Concurrent Code with ConTest

1. Remove non-relevant code (optional)
2. Create a number of tests
3. Apply ConTest to the code
4. Run each test multiple time and measure synchronization coverage
   1. If a bug is found, use ConTest to debug, fix, go to 3
   2. If deadlock violation found, fix, go to 3
5. If coverage not sufficient, analyze
   1. Need to rerun tests, go to 4
   2. Need to write new tests, go to 2
6. Done
Function and System Test with ConTest

- Increased efficiency in finding intermittent bugs
- Negligible change to process in Java (in C recompilation is required)
  - Installed in WebSphere size projects in a day
  - Works under any test automation used (e.g., Rational tools)
- Additional benefits:
  - Coverage measurements
  - Aids in debugging
  - Replay feature
Measuring Concurrent Coverage

- ConTest also measures coverage
  - The feature that people wanted most is not to find bugs 😞

- How do you know if the tests are any good from concurrency view?

- Synchronization coverage
  - Make sure that every synchronization primitive did something

- See - Applications of synchronization coverage
Finding Potential Deadlocks

- Standard idea – find violation of lock discipline
- Our innovation – enable cross run collection of data
- Attention is being paid to it in MSR
Automatic Debugging as Feature Selection

- Think of each instrumentation point as a feature
- Execute the program many times with different subset
  - Each instrumentation point gets a score
  - $P(i) = \frac{P(\text{success}|X_i)}{P(\text{!success}|X_i)}$

- There are false negatives and possessives

- The nodes with the highest score may be related to location of failure
Score Distributed Along Program Lines
The First Largish Program We Tried It On
This Does Not work When The Problem is Easy
Looking at the Differences on the First Large Program
Looking on the Differences at the Second Large Program
Healing Concurrent Programs

- In testing we try to make them more likely to fail
- In the field we may want to make them less likely to fail

- General problem statement – the program sometimes fail (depends on the interleaving)
  - General solution – eliminate, or make the interleaving the lead to bugs less likely; but do not cause deadlocks

- We do it for races, deadlocks …
  - Not quite as useful but very interesting
  - Done as a project called SHADOWS in the EU