A Classification of Concurrency Failures in Java components

Brad Long, Paul Strooper
The University of Queensland, Australia

Overview

• Issues in testing concurrent java components
• Solution for testing concurrency
  – A Model of Java concurrency
  – A Classification of Concurrency Failures
  – Applying the Model to Test case selection
    • Producer-consumer example
• Conclusion

Concurrent Programs Testing Issues

• Concurrent programs are hard to test due to their inherent non-determinism.
  – That is even if we run a concurrent program twice with the same input, it is not guaranteed to return the same output both times.
• Concurrency leads to many interleavings and non-determinism
  – test cases may have to be run multiple times
  – traditional notions of test coverage may not be sufficient
  – automated checking of outputs may be difficult

Concurrent Components

• Concentrating on the concurrent components
  – Complexity of testing concurrent programs is significantly reduced by focussing on concurrent components rather then the entire system
• A component is a unit of composition with contractually specified interfaces and explicit context dependencies (Szymerski, 1998)
  – typically one or more Java classes (monitors)
  – assume it will be accessed by multiple threads
  – such a component is likely to come to life through objects and therefore would normally consist of one or more classes.
Java primitives

- The key Java primitives for thread synchronization are:
  - `synchronized` methods and blocks;
  - the methods `wait`, `notify` and `notifyAll` from the `Object` superclass.

- `Thread creation, join, sleep, and interrupt` are not discussed since these are not typically found in concurrent components themselves, but in the multithreaded programs that use these components.

- The methods `suspend`, `resume` and `stop` are also not discussed because they are deprecated and their use is discouraged.

Java Primitives - Mutual exclusion in Java

- **Mutual exclusion** is achieved by a thread locking an object. Two threads cannot lock the same object at the same time, thus providing mutual exclusion.

- A thread that cannot access a synchronized block because the object is locked by another thread is **blocked**.

- In Java there are two ways of locking an object.

Explicitly call a synchronized block.

```java
synchronized (anObject) {
    ...
}
```

Synchronize a method.

```java
public synchronized void aMethod() {
    ...
}
```

Or

```java
public void aMethod() {
    synchronized (this) {
        ...
    }
}
```

Java Primitives - Waiting and Notification

- **Suspended thread** - Threads are suspended by calling the Java `wait` method. This causes the lock on the object to be released, allowing other threads to obtain a lock on the object. Suspended threads remain dormant until woken.

```java
public synchronized Item get() {
    while (buffer.size() == 0) {
        wait();
    }
    ...
}
```

```java
public synchronized void put(Item item) {
    ...
    buffer.add(item);
    notify();
}
```

Systematic Testing of a Concurrent Program

- Write the Model
  - Petri-nets are used to represent the model in a graphical manner.

- Classify concurrency failures
  - Model is used to classify the failures that can occur in concurrent Java components and determine suitable tools and techniques for each class of failure.

- Draw Concurrency Flow graphs for each method

- Create Test sequence from traces
  - The test sequences can be used to construct test drivers or as input to dynamic analysis testing tools (ConAn).
  - Execute test sequences in ConAn. ConAn used to execute test sequences and evaluate outputs.
Step 1: The Model of Java Concurrency

- **Petri-nets Model**
  - represents the states and transitions of a single thread with respect to a synchronized object at any point of time.
  - provide a convenient mechanism for modeling the locking of objects.
- This representation has been chosen to highlight two issues
  - change in state of a thread when concurrent constructs are encountered in a multithreaded program, and
  - the effect that availability of the object lock has on a thread’s state.

**Petri-Net specification**
- Consists of four types of components: places (circles), transitions (rectangles), arcs (arrows) and markers/tokens (dots).
  - Places represent possible states of the system;
  - Transitions are events or actions which cause the change of state; And
  - Every arc simply connects a place with a transition or a transition with a place.
  - Tokens can represent: resource availability, jobs to perform, flow of control, synchronization conditions ...
- Change of status is denoted by a movement of markers/token(s) (black dots) from place(s) to place(s); and is caused by the firing of a transition.
- The firing represents an occurrence of the event or an action taken.

**Petri-Net model of Concurrency**

**Transition T1: Requesting an Object Lock**
- Transition T1 is fired by a thread entering a synchronized block.
- A marker exists in place A, therefore transition T1 can fire causing the marker to move to B.
- Place B represents a thread requesting an object lock.
**Transition T2: Locking an Object**

- Transition T2 is fired by the JVM serving the requesting thread an object lock. If an object lock is available, that is, if a marker exists in place E, the marker can move to C.

- Place C represents a thread executing in a critical section with the object lock. If no lock is available, the thread is blocked in B.

**Transition T3: Waiting on an Object**

- Transition T3 represents a thread entering the wait state.

- This occurs when the code calls the wait method, which also releases the object lock, hence the arc to place E.

- From C, a marker is moved to both D and E.
**Petri-Net Model after Transition T3 fired**

- **A** - outside a synchronized block
- **B** - requesting entry to a critical section
- **C** - executing in a critical section
- **D** - wait state
- **E** - object lock is available

**Transitions**
- **T1** - fired by thread entering a sync block
- **T2** - fired by JVM serving the thread an object lock
- **T3** - fired by thread entering the wait state
- **T4** - fired by thread leaving the sync block
- **T5** - waiting thread waking up

**Arc**
- Transition T4: Releasing an Object Lock
  - Transition T4 represents a thread leaving a synchronized block.
  - When this occurs, a marker is placed in both A and E.
  - This transition causes the lock on the object to be released.

**Petri-Net Model after Transition T4 fired**

- **A** - outside a synchronized block
- **B** - requesting entry to a critical section
- **C** - executing in a critical section
- **D** - wait state
- **E** - object lock is available

**Transitions**
- **T1** - fired by thread entering a sync block
- **T2** - fired by JVM serving the thread an object lock
- **T3** - fired by thread entering the wait state
- **T4** - fired by thread leaving the sync block
- **T5** - waiting thread waking up

**Arc**
- Transition T5: Thread Notification
  - Transition T5 represents a waiting thread waking up.
  - When this occurs, the marker moves to B to reacquire the object lock it was waiting on.
  - The incoming dashed arc at T5 represents another thread notifying the waiting thread. This has the obvious implication that a thread in the wait state cannot wake itself.
Step 2: A Classification of Concurrency Failures

- Using a HAZOP style of analysis, we analyze each transition for two deviations,
  - 1) failure to fire the transition, and
  - 2) erroneous firing of the transition.
- This approach is taken for completeness, to ensure all failures are identified and classified.
- The correct transition firings plus the two deviations form a complete set of transition firings.

A Classification of Concurrency Failures – Results of Analysis

- **Transition**: the name of the transition under analysis
- **Failure**: a categorization of the failure. Two classifications, *failure to fire* and *erroneous firing*, are used.
- **Cause**: a brief description of possible causes of the failure.
- **Conditions**: the conditions under which the failure can occur.
- **Consequences**: the consequences of the failure.
- **Testing Notes**: any notes relating to testing implications. Generally a method or approach for detecting the failure is listed (Static/dynamic/Model checking/Check call completion time).

A Classification of Concurrency Failures – Testing Notes

- **Static Analysis** – involves the analysis of a program without requiring execution. Typically this involves the generation and analysis of models of states and transitions of a program.
- **Static and Dynamic Analysis** – Some tools combine static and dynamic analysis. For example, JPF’s runtime analysis utilizes the LockTree and Eraser algorithms for detecting potential deadlocks and race conditions. The static analysis phase collects information to allow the more accurate dynamic phase to execute efficiently.
- **Model Checking** – This involves the formal analysis and mechanical checking of software systems, thus avoiding the tedium and introduction of errors inherent in manual formal methods. Approaches based on model checking includes Bandera, JPF, JLint.
- **Deterministic testing** – Requires a forced execution of the program according to an input test sequence. Ex: *Check call completion time (ConAn)*.
A Classification of Concurrency Failures – Check call Completion Time

- Check call completion time
  - This technique uses deterministic execution to allow a tester to specify sequences of method calls. To guarantee the order of execution, the method uses an abstract clock to provide synchronization.
- This clock provides three operations:
  - `await(t)` delays the calling thread until the clock reaches time t,
  - `tick` advances the time by one unit, waking up any processes that are awaiting that time,
  - `time` returns the number of units of time passed since the clock started.
    - The `time` operation is added to detect when threads wake up.
- `ConAn` automates these steps by allowing the tester to specify the sequence of monitor calls and by assigning each call a thread.

Concurrent Failure Classification – Transition T1 failures

<table>
<thead>
<tr>
<th>Transition</th>
<th>Failure</th>
<th>Cause</th>
<th>Conditions</th>
<th>Consequences</th>
<th>Testing notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Failure to fire T1</td>
<td>Thread does not access a</td>
<td>Two or more threads access a</td>
<td>Interference (also known as a race condition or data race)</td>
<td>Static analysis/model checking (often combined with dynamic analysis)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>synchronized block when</td>
<td>Shared resource</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>required</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Erroneous firing of T1</td>
<td>Program logic accesses critical</td>
<td>No more than one thread accesses</td>
<td>Unnecessary synchronizatio</td>
<td>Static analysis/model checking (often combined with dynamic analysis)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>section.</td>
<td>shared resources. The thread is</td>
<td>n.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>not required to wait or notify</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>other threads.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concurrent Failure Classification – Transition T2 failures

<table>
<thead>
<tr>
<th>Transition</th>
<th>Failure</th>
<th>Cause</th>
<th>Conditions</th>
<th>Consequences</th>
<th>Testing notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>Failure to fire T2</td>
<td>The object lock to be acquired</td>
<td>Another thread has acquired the</td>
<td>The thread is</td>
<td>Static analysis and dynamic analysis.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>has been acquired by another</td>
<td>lock being</td>
<td>permanently</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread.</td>
<td>acquired by this thread. This</td>
<td>suspended</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>can occur in two ways</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1)One thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Continuously holds the lock.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2)One or more threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>repeatedly acquire the lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>being requested by this thread.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Erroneous firing of T2</td>
<td>Not applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concurrent Failure Classification – Transition T3 failures

<table>
<thead>
<tr>
<th>Transition</th>
<th>Failure</th>
<th>Cause</th>
<th>Conditions</th>
<th>Consequences</th>
<th>Testing notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td>Failure to fire T3</td>
<td>No call to wait is made</td>
<td>Thread is required to make a</td>
<td>Program code may</td>
<td>Check completion time call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>call to wait</td>
<td>erroneously execute in</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the critical section,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>or leave critical section</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>permanently.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Erroneous firing of T3</td>
<td>Program logic makes an erroneous</td>
<td>A call to wait is not desired</td>
<td>A thread may suspend</td>
<td>Check completion time call</td>
</tr>
<tr>
<td></td>
<td></td>
<td>call to wait</td>
<td></td>
<td>indefinitely if no</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>other thread exists to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>notify it. The object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lock is released.</td>
<td></td>
</tr>
</tbody>
</table>
Concurrent Failure Classification – Transition T4 failures

<table>
<thead>
<tr>
<th>Transition</th>
<th>Failure</th>
<th>Cause</th>
<th>Conditions</th>
<th>Consequences</th>
<th>Testing notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4</td>
<td>Failure to fire T4</td>
<td>The thread never releases the object lock.</td>
<td>Thread is either in endless loop, waiting for blocking input (which is never received), or acquiring an additional lock which is locked by another thread</td>
<td>Thread never completes. Other thread may be blocked if they are waiting for the lock.</td>
<td>Check completion time call</td>
</tr>
<tr>
<td></td>
<td>The thread fires T3, that is it waits instead.</td>
<td>none</td>
<td>Thread waits instead of completing and leaving the critical section</td>
<td></td>
<td>Check completion time call</td>
</tr>
<tr>
<td></td>
<td>Erroneous firing of T4</td>
<td>The thread releases the object lock prematurely</td>
<td>none</td>
<td>Thread exists and subsequent statements may access shared resource.</td>
<td>Check completion time call</td>
</tr>
</tbody>
</table>

Concurrent Failure Classification – Transition T5 failures

<table>
<thead>
<tr>
<th>Transition</th>
<th>Failure</th>
<th>Cause</th>
<th>Conditions</th>
<th>Consequences</th>
<th>Testing notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5</td>
<td>Failure to fire T5</td>
<td>Thread is not notified</td>
<td>No other thread calls notify whilst this thread is in wait state.</td>
<td>Thread is permanently suspended.</td>
<td>Check completion time call</td>
</tr>
<tr>
<td></td>
<td>(waiting thread waking up)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Erroneous firing of T5</td>
<td>Thread is notified</td>
<td>None</td>
<td>Thread prematurely reenters the critical section.</td>
<td>Check completion time call</td>
</tr>
</tbody>
</table>

Step3 : Applying Model to Test Case Selection

- Extend Brinch Hansen approach for testing Concurrent monitors. This method provides a systematic method for testing concurrent components. Method consisting of four steps:
  1. Identify set of preconditions that exercise each monitor method in desired way.
  2. Construct a set of test sequences of monitor calls to satisfy the test conditions identified in step 1.
  3. Construct a test driver that starts a number of threads that call the component in the order prescribed in step 2.
  4. Execute test program and compare output to expected output.
- Systematic white-box approach is used for test-case selection based on the model and classification of concurrency failures (illustrate the approach with a producer and consumer monitor).
- Steps 3 and 4 have been automated with ConAn.

An Example: Producer consumer monitor

```java
class ProducerConsumer {
    String contents; // String of characters
    int totalLength, curPos = 0;

    // receive a single character
    public synchronized char receive() {
        char y;
        // wait if no character is available
        while (curPos == 0) {
            wait();
        }
        // retrieve character
        y = contents.charAt(totalLength-curPos);
        curPos = curPos - 1;
        // notify blocked send/receive calls
        notifyAll();
        return y;
    }
    // end of receive
}
```
An Example: Producer consumer monitor (cont..)

```java
// send a string of characters
public synchronized void send(String x) {
    // wait if there are more characters
    while (curPos > 0) {
        wait();
    }
    // store string
    contents = x;
    totalLength = x.length();
    curPos = totalLength;
    // notify blocked send/receive calls
    notifyAll();
}
} //end of ProducerConsumer
```

Concurrent Flow Graph (CoFG)

- **CoFG**
  - To achieve coverage of all concurrent statements
  - Constructions is straight forward
  - contains all statements that cause transitions as described in our model. That is identify the code regions between all pairs of concurrent statements in each method.
  - Each arc in the graph is a unique.

- **Build test sequences that exercise arcs of the CoFGs.**
  - This involves creating a test driver that instantiates a number of threads which make calls on the synchronized methods.
    The test driver can easily be created by using the ConAn concurrency testing tool.
  - The sequence of calls should ensure coverage of the CoFGs.

CoFG for Producer and Consumer

- **Start -> wait**
  - This represents the following transition firings from our model: T1, T2, T3.

- **Wait -> wait**
  - This covers the transition firings T3, T5, T2, T3.

- **Wait -> notifyAll**
  - Transitions fired: T3, T4, T5.

- **Start->notifyAll**
  - Transitions fired: T1, T2, T5.

CoFG for receive methods

- **notifyAll -> end**
  - Transitions fired: T5, T4
1. Identify Test Conditions

**send**
- C₁ Start – wait
- C₂ wait – wait
- C₃ wait - notifyall
- C₄ start – notifyall
- C₅ notifyall - end

**receive**
- C₆ Start – wait
- C₇ wait – wait
- C₈ wait - notifyall
- C₉ start – notifyall
- C₁₀ notifyall - end

```java
public synchronized void send(Object o) {
    while (/* if there are more characters */)
        wait();
    /* add item to buffer */
    notifyAll();
}
```

```java
public synchronized void receive() {
    while (/* if no character available */)
        wait();
    /* retrieve character */
    notifyAll();
}
```

2. Construct Test Sequences

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread</th>
<th>Call</th>
<th>Output</th>
<th>Conditions</th>
<th>Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Sender 1</td>
<td>send(“a”)</td>
<td>–</td>
<td>C₄</td>
<td>–</td>
</tr>
<tr>
<td>T2</td>
<td>Sender 2</td>
<td>send(“b”)</td>
<td>–</td>
<td>C₁</td>
<td>Sender 2(T2)</td>
</tr>
<tr>
<td>T3</td>
<td>Receiver 1</td>
<td>Receiver()</td>
<td>“a”</td>
<td>C₉</td>
<td>–</td>
</tr>
<tr>
<td>T4</td>
<td>Receiver 2</td>
<td>Receiver()</td>
<td>“b”</td>
<td>C₉</td>
<td>–</td>
</tr>
</tbody>
</table>

3. Implement Sequences in Driver

```java
#begin_case
#goal_conditions C₁ C₄ C₉
#begin_tick // T1
#begin_thread sender 1
    #excMonitor m.send("a"); #end
    #valueCheck time() # 1 #end
#end_thread
#end_tick

# begin_tick // T2
    #begin_thread sender 2
        #excMonitor m.send("b"); #end
        #valueCheck time() # 3 #end
    #end_thread
#end_tick

# begin_tick // T3
    #begin_thread receiver 1
        #valueCheck m.receiver(); # 'a' #end
        #valueCheck time() # 3 #end
    #end_thread
#end_tick

# begin_tick // T4
    #begin_thread receiver 2
        #valueCheck m.receiver(); # 'b' #end
        #valueCheck time() # 4 #end
    #end_thread
#end_tick
#end_case
```

4. Execute Test Driver

![Test Driver Generation](image1)

**TEST DRIVER GENERATION**

**TEST DRIVER EXECUTION**

***** Test cases: 84
***** Value errors: 0
***** Exception errors: 0
***** Liveness errors: 0
ConAn Features & Limitations

• ConAn features
  – reduces testing of concurrent components to something familiar
  – allows for testing of non-deterministic output
  – detects liveness errors

• Limitations
  – tester must define test conditions and test sequences
  – difficult to detect problems with interference
  – can control some non-determinism, but not all
    • no control over order in which JVM grants locks
    • no control over order in which JVM removes threads from wait set

Conclusion

• Complexity is significantly reduced by focusing on concurrent components rather than entire systems.
• A component is tested under the assumption of multiple thread access.
• The classification for concurrency failures provides us with a motivation for a test case selection strategy using concurrency flow graphs.
• It potentially removes the need for white-box techniques.
• In addition, the classification highlights the importance of checking thread completion times since this can be used in many cases to detect transition failures.
• By applying this technique in combination with black-box testing, we believe a superior technique can be devised.