Testing of Parallel Programs

Dr. Christoph Steindl

Why is Concurrent Testing Hard?

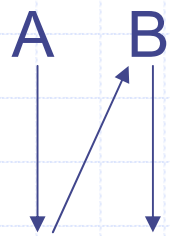
- Concurrency introduces non-determinism
  - Multiple executions of the same test may have different interleaving (and different results!)
  - Re-executing a test on a single stand-alone processor is not useful
- Debugging affects the timing
- No useful coverage measures for the interleaving space
- Result: Most bugs are found in system tests, stress tests, or by the customer

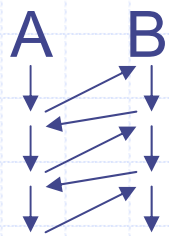
Interleaving

- Interleaving is the relative execution order of program threads.
- Threads A and B execute inc().
- The result depends on the interleaving.
- This would not be revealed in a typical test.

```
1.  public class Interleaving {
2.    static int global = 0;

3.    public static void inc() {
4.      int temp = global;
5.      temp = temp + 1;
6.      global = temp;
7.    }
8.  }
```





Result: global = 2 global = 1

Java Source Code and Bytecodes

```
public void inc() {
   Global += Local;
}
```

```
Method void inc()
  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
```

Why are these bugs not found?

- **Frame of mind when the program is written**
- **Requires thread switching at precise locations**
- **Typical testing environment**
  - Thread switch occurs at repeating locations
  - Execution is almost deterministic
  - No load/stress
- **Not enough tests**
- **Not enough of the right kind of tests**

Testing Parallel Programs with ConTest

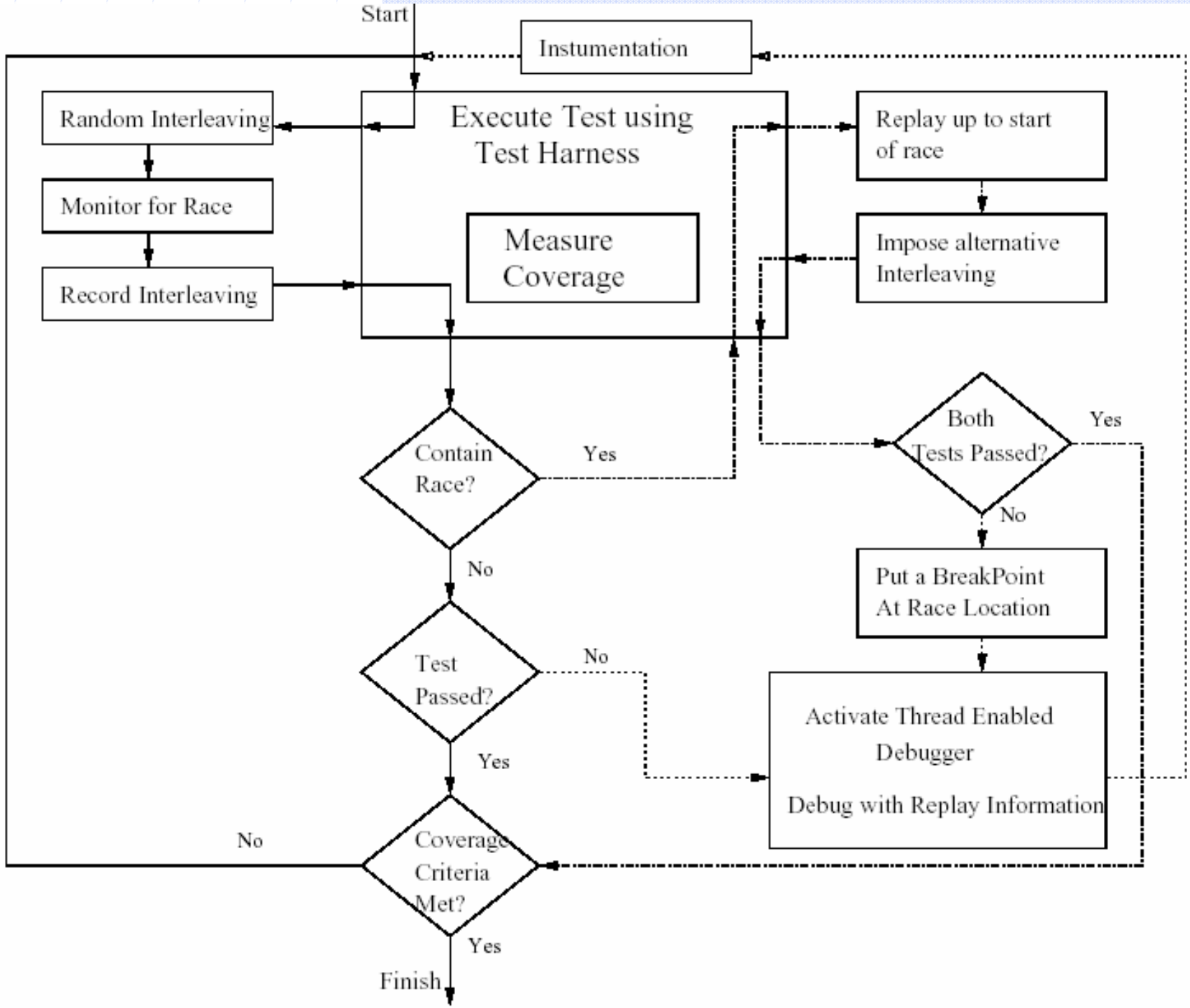
- Develop the program just as any other program (e.g. with TDD).
- Start with your automated unit tests.
- How to use ConTest?
  - Instrument the program with ConTest.
  - Re-execute the tests (with the ConTest library on the classpath)
  - Verify that the tests still pass (or correct any errors).
  - Use coverage information to add tests for parallelism.

How Does ConTest-Lite Find Bugs?

- **ConTest instruments every concurrent event**
  - Concurrent events are the events whose order determines the result of the program (a synchronization primitive like a “synchronized block” or an “object.wait” or shared memory access)
  - At the bytecode level
  - Creates hooks for the irritator and for coverage printing
  - Generates coverage models
  - Instrumentation can be limited to selected parts
- **At every concurrent event, a random based decision is made whether to cause a context switch**
  - For example, using a sleep statement
- **Philosophy:**
  - Modify the program in such a way that it will be more likely to **exhibit** bugs (without introducing new bugs)
  - Minimize impact on the testing process
  - Re-use existing tests
  - Utilize the time computers are not being used (nights, weekends, etc.)

Race Detection

- A race is usually defined as two accesses to the same memory, at least one of which is a write, done by two different threads with no synchronization between the accesses.
- Unlike all other race detection tools, in ConTest the race detection component *never* reports on races to the user.
- When this component finds a race, it communicates with the replay and irritator components to ensure that the test is re-executed; this time the race will be forced to occur in the opposite order.
- If the race results in a bug, the user can view any execution that caused the bug with a debugger, and to stop at a breakpoint just before the race occurs.


Start
Instumentation
Random Interleaving
Execute Test using
Test Harness
Replay up to start
of race
Monitor for Race
Measure
Coverage
Impose alternative
Interleaving
Record Interleaving
Contain
Race?
Yes
Both
Tests Passed?
Yes
No
No
Put a BreakPoint
At Race Location
Test
Passed?
No
Activate Thread Enabled
Debugger
Debug with Replay Information
Yes
No
Coverage
Criteria
Met?
Yes
Finish

Instrumentation in detail

- Every read of variable v is replaced by:
  - (type of v) after_read(before_read(id), v, id)
- Any part of the expression which represents a write, v = SomeExpression, is replaced by:
  - (type of v)after_write(v = (type of v) before_write(SomeExpression, id), id)

Concurrent Coverage Model

- Synchronized block
  - **synchronization visited** ➔ covered if this synchronization block was reached
  - **synchronization blocking** and **synchronization blocked**
    ➔ covered when a thread reached a synchronized block A, and stopped because another thread was inside a block B synchronized on the same lock.
    In this case, block A will be reported as blocked, and block B as blocking (both in addition to visited).
  - Some synchronization tasks cannot be covered - for example, if a synchronized block is necessarily the first to be performed in the program, it can only be blocking, and never blocked. But this is rare.
  - Normally, a synchronization block can sometimes block and sometimes be blocked. If you don't get a full coverage, this is a cause for concern: the test may be insufficient, or alternatively (if there can't possibly be contention) the synchronization may be redundant.
  - **synchronization retaking**
    ➔ reported when a thread synchronizes on a lock it is already holding (that is, at the inner block). This type does not appear in the list of tasks to be covered, but may be seen when viewing the runtime coverage trace. If a synchronization was fully covered (was both blocking and blocked), it doesn't matter whether it was also retaking or not. But if a task is missing - a synchronization was never blocked or never blocking - checking whether it was retaking may help in understanding why.
- Object.wait call
  - **wait visited** ➔ covered if this synchronization block was reached
  - **wait repeated** ➔reported if this wait was called twice within the same synchronized block

Other types of Concurrent Coverage

- Shared variables coverage
  - Names of variables detected as shared in a given run, i.e accessed by more than one thread, written to the directory sharedVarTraces.
  - Each line in the trace files describes one variable. It contains the full class name and the variable name, separated by space. For example, com.ibm.some_project.SomeClass someMember.
- Interfered location pairs coverage
  - writes its files to directory locationPairsTraces.
  - Each line in the trace file of this coverage type contains a pair of program locations that were encountered consecutively in the run, and a third field which is "t" or "f". It is "f" if the two locations were run by the same thread, and "t" otherwise - that is, "t" means there was a context switch there.
  - It can be used to test the quality of the tests, and whether we actually get interleavings we didn't get before - whether we have context switches in new places.

References

- Shmuel Ur, http://cs.haifa.ac.il/courses/softtest/testing2004/index.html
- Multithreaded Java program test generation, IBM Systems Journal, Vol. 41, no. 1, 2002, „Software Testing and Verification“