Testing and Debugging Concurrent Software – Challenges and Solutions

Shmuel Ur

Eliminating Defects from Concurrent Software

- **Fact**

  Concurrency is pervasive
  - Infrastructures
  - Middleware
  - Desktop
  - Embedded software
  - High performance

Northeastern blackout, 8/14/03

Concurrent software bug

- **Consequence**

  Fundamental need to use powerful testing tools for concurrent software

Why is Concurrent Testing Hard?

- Concurrency introduces **non-determinism**
  - Multiple executions of the same test may have different interleavings and different results
  - 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 unproportional 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

```
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

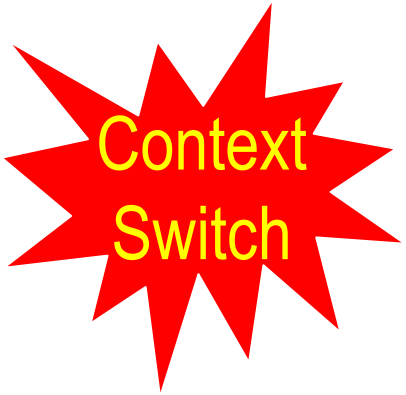
...

load operand

increment

write-back

...



Thread B

...

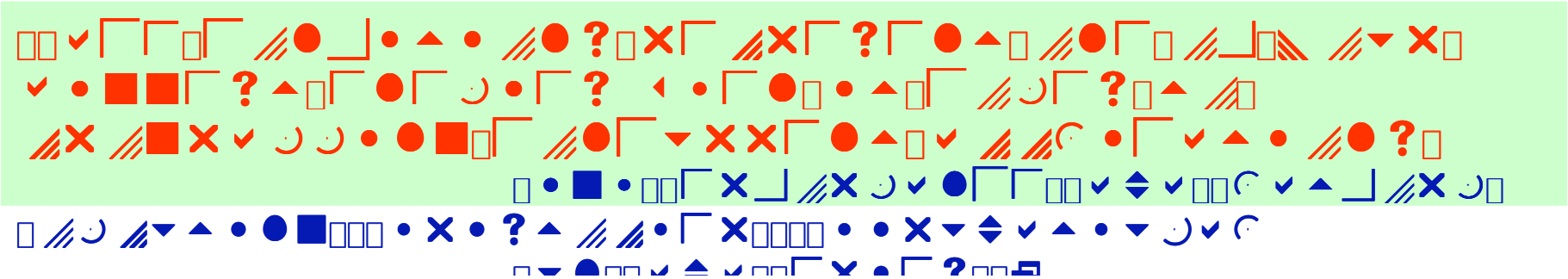
load operand

increment

write-back

...

counter = 4



Riddle I: How many bugs do you see?

```
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("\n");
   }
}
```

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

```
class World extends Thread{
    public void run(){
        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?

Riddle III: How Much is 1+10+100+1000+10000?

```
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 :

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

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

ThreadA	ThreadB	ThreadC	ThreadD	ThreadE	TOTAL COVERA...	UNIQUE COVE...	OUT OF	percentage
0	1	0	0	1	0	0	1	0.0
0	1	0	0	0	0	0	1	0.0
0	0	1	0	1	0	0	1	0.0
1	0	0	1	1	0	0	1	0.0
0	0	1	0	0	0	0	1	0.0
1	0	0	1	0	0	0	1	0.0
1	1	1	1	1	1000	1	1	100.0
1	1	1	1	0	0	0	1	0.0
0	1	1	0	1	0	0	1	0.0
0	1	1	0	0	0	0	1	0.0
1	0	1	1	1	0	0	1	0.0
1	0	1	1	0	0	0	1	0.0
0	0	0	1	1	0	0	1	0.0
0	0	0	1	0	0	0	1	0.0
1	1	0	0	1	0	0	1	0.0
1	1	0	0	0	0	0	1	0.0
0	1	0	1	1	0	0	1	0.0
0	1	0	1	0	0	0	1	0.0
0	0	1	1	1	0	0	1	0.0
0	0	1	1	0	0	0	1	0.0
1	0	0	0	1	0	0	1	0.0
1	0	0	0	0	0	0	1	0.0
1	1	1	0	1	0	0	1	0.0
1	1	1	0	0	0	0	1	0.0
0	1	1	1	1	0	0	1	0.0
0	1	1	1	0	0	0	1	0.0
1	0	1	0	1	0	0	1	0.0
1	0	1	0	0	0	0	1	0.0

Print Export Close

Coverage Results with ConTest

Here are the legal tasks :

Number of Tasks: 32 Covered: 32 Coverage Percentage: 100.0

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

ThreadA	ThreadB	ThreadC	ThreadD	ThreadE	TOTAL COVERAGE	UNIQUE COVERAGE	OUT OF	percentage
0	0	0	0	0	2	1	1	100.0
1	0	0	0	0	20	1	1	100.0
0	1	0	0	0	18	1	1	100.0
1	1	0	0	0	36	1	1	100.0
0	0	1	0	0	5	1	1	100.0
1	0	1	0	0	47	1	1	100.0
0	1	1	0	0	15	1	1	100.0
1	1	1	0	0	68	1	1	100.0
0	0	0	1	0	3	1	1	100.0
1	0	0	1	0	34	1	1	100.0
0	1	0	1	0	19	1	1	100.0
1	1	0	1	0	62	1	1	100.0
0	0	1	1	0	5	1	1	100.0
1	0	1	1	0	66	1	1	100.0
0	1	1	1	0	19	1	1	100.0
1	1	1	1	0	100	1	1	100.0
0	0	0	0	1	2	1	1	100.0
1	0	0	0	1	17	1	1	100.0
0	1	0	0	1	13	1	1	100.0
1	1	0	0	1	32	1	1	100.0
0	0	1	0	1	4	1	1	100.0
1	0	1	0	1	57	1	1	100.0
0	1	1	0	1	11	1	1	100.0
1	1	1	0	1	45	1	1	100.0
0	0	0	1	1	3	1	1	100.0
1	0	0	1	1	21	1	1	100.0
0	1	0	1	1	17	1	1	100.0
1	1	0	1	1	60	1	1	100.0
0	0	1	1	1	10	1	1	100.0
1	0	1	1	1	67	1	1	100.0
0	1	1	1	1	15	1	1	100.0
1	1	1	1	1	107	1	1	100.0

Print Export Close

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.

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...**

Typical Concurrent Bug Patterns

Non-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++

Atomicity is Never Ensured

```
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

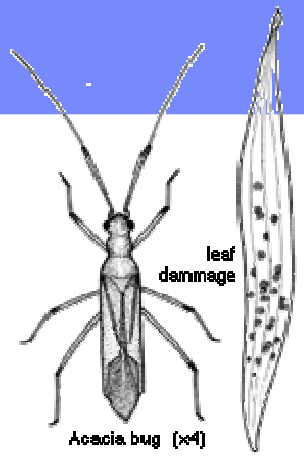
Possible bug:

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?

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

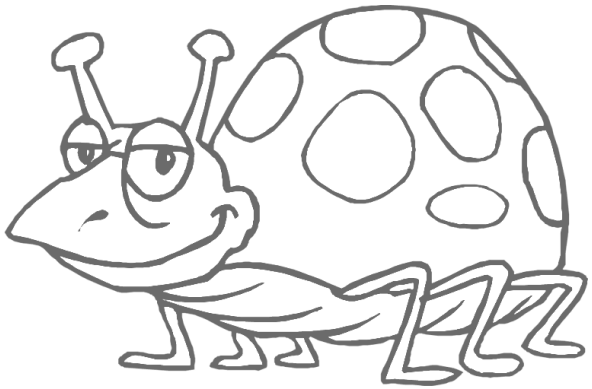
- Protection of thread one does not apply to thread two
- There is an access protocol that is not followed due to:
  - A new team member
  - An attempt to improve performance

<u>Thread 1</u>

```
Synchronized (o){
  x++;
}
```

<u>Thread 2</u>

```
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

```
Synchronized (o){
  o.wait();
}
```

Thread 2

```
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
- Is actually in a different thread
- Can result in concurrency bugs

Nasty to have a hanging 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:
- synchronized (obj) {
  - while (<condition does not hold>)
    - obj.wait(timeout); ... // Perform action appropriate to condition
  - }

Visibility

- When a heap variable is updated
  - It is updated in the registers or “thread local memory”
  - It reaches the heap on specific language dependant events
    - For example, a lock release
  - The programmer assumes that a condition updated by one thread is visible to another thread—when it is not

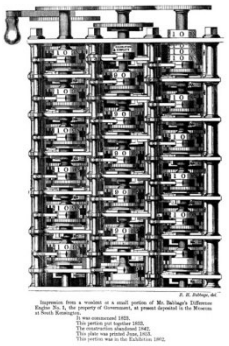
Some Interesting Observations

- 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!

Types of Technologies used to Test Multi-threading

- Formal verification
- Static analysis
- State space exploration
- Race detection
- Trace analysis
- Replay
- Cloning
- Noise makers

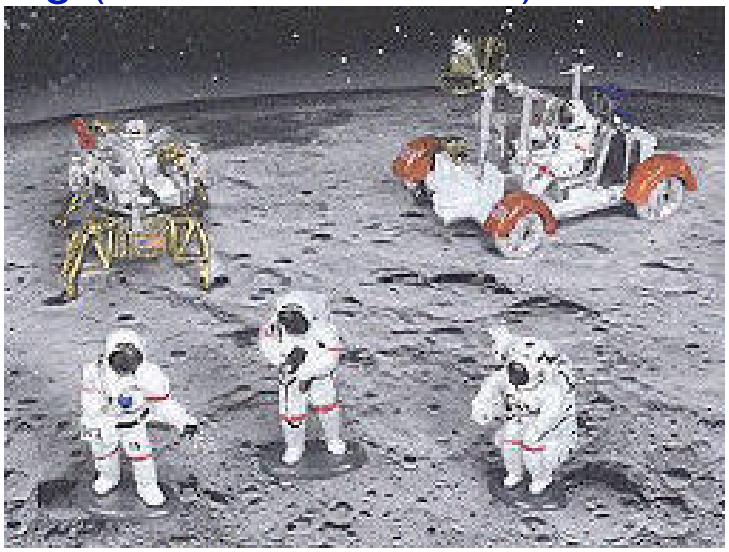
Formal Verification and Static Analysis



- Analyzing the code and looking for bugs
  - Model checkers usually verify invariants
  - Detect risky programming practices
- Constructing models for verification
  - Static analyzers detect relevant parts of the program; however, they tend to be either too cautious or too optimistic
  - Formal verification is limited by the state explosion problem
- Lots of research – practical impact only in limited domains
  - Tools: FeaVer, Bandera, SLAM, BLAST, TVLA, Canvas
  - Papers in many conferences

State Space Exploration

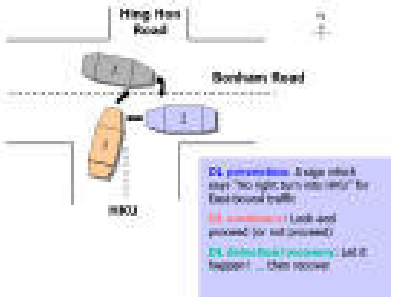
- Integrates automatic test generation, execution, and evaluation
- Run random test
  - Expected results are not known
- Generate random inputs and random timing (or biased random)
- Try to create
  - Deadlocks
  - Exceptions
  - Violation of user-defined assertions
- Try to explore all program states
- Tools: Verisoft, CMC, JPF



Race Detectors

- A race is two accesses to the same variable, from two different threads that are not synchronized, at least one of which is a write
- Race detectors can work on-line or off-line
  - On-line means lower performance
  - Off-line requires large traces

- Not all information reported is correct
  - The race could be intentional
  - The synchronization may be subtle or user implemented
- A lot of research in this domain, mainly on improving performance and accuracy



Trace Analyzers

- Post mortem of the execution
- Can check if properties hold
- Can look for races

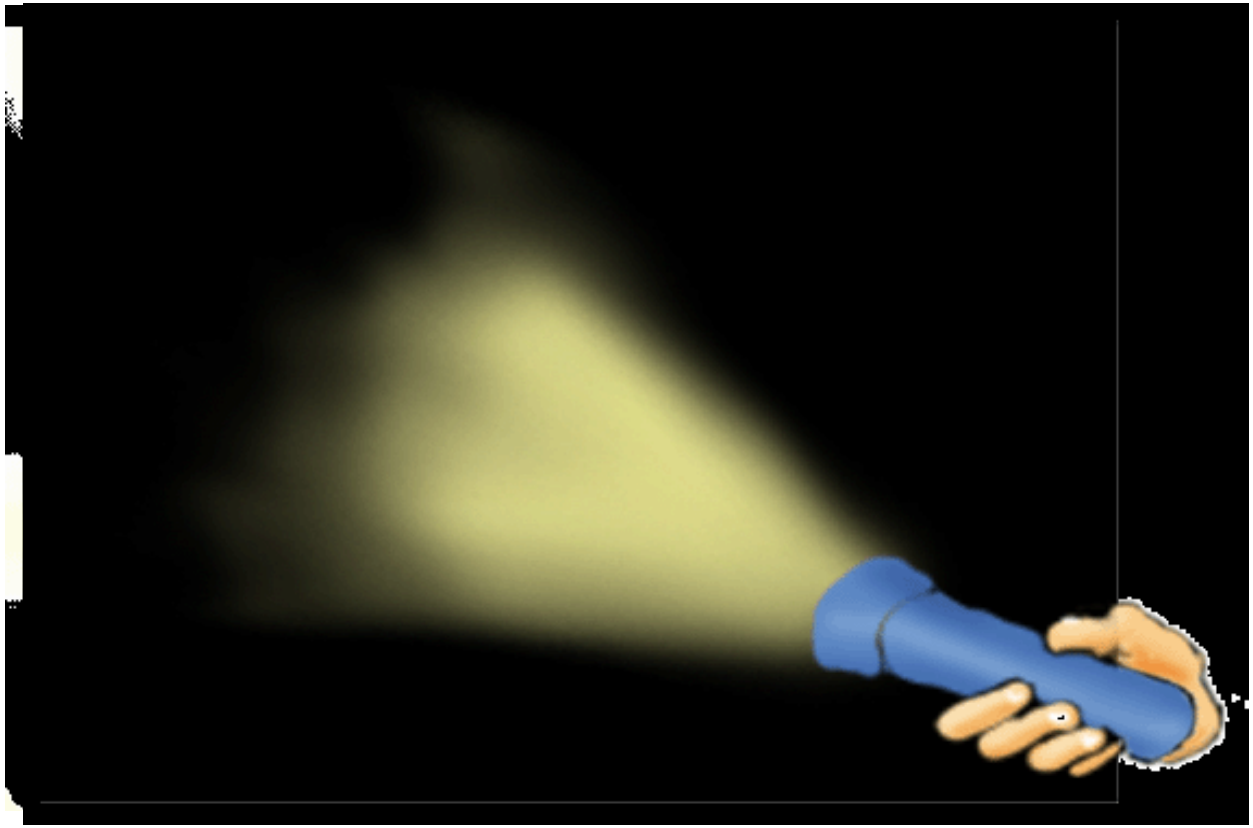
Replay

- Once a bug is found you want to be able to replay the test
- Accurate replay is difficult
  - All OS interaction has to be captured
  - All application randomness (time, random, hash) captured
  - All input captured
  - Interleaving captured
- Seed replay is much easier
  - Repeat the randomness caused by the application
- Replay tool may be at the source, bytecode, or JVM level

Cloning

- Run the “same” test multiple times in parallel
  - Load testing – the most used testing technology is based on this observation: LoadRunner, Robot…
  - Contention is likely
  - Usually limited expected results are used
  - Need to distinguish between clones

Noise Makers



With ConTest each Test Goes a Long Way

How Does ConTest Find Bugs?

- ConTest instruments every concurrent event
  - Concurrent events are the events whose order determines the result of the program, such as accesses to shared variables, calls to synchronization primitives
- At every concurrent event, a random-based decision is made whether to cause a context switch by injecting “noise”
  - E.g., using a sleep statement
- Philosophy
  - Modify the program so it is more likely to exhibit bugs (without introducing new bugs – no false alarms)
  - Minimize impact on the testing process (under-the-hood technology)
  - Reuse existing tests

Why ConTest is needed

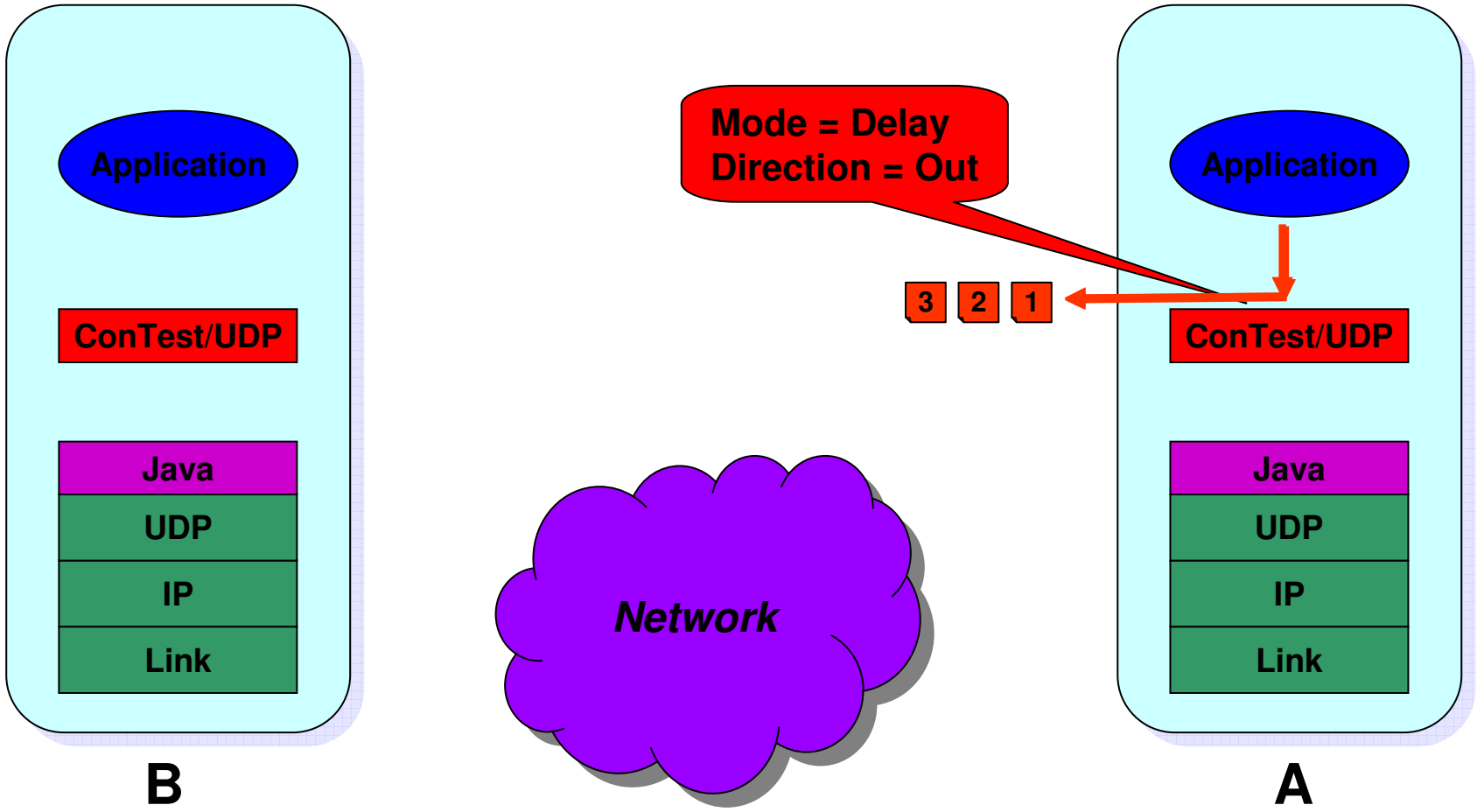
- Most unit tests which test concurrency, don't *really* test concurrency.
- As an exercise, remove the synchronization protocol from the code, and run the test.
  - From our experience, most probably the test will run just as well…
- Something is needed in order to make contention actually happen.
- ConTest's noise injection is that something.

- Download from: http://www.alphaworks.ibm.com/tech/contest

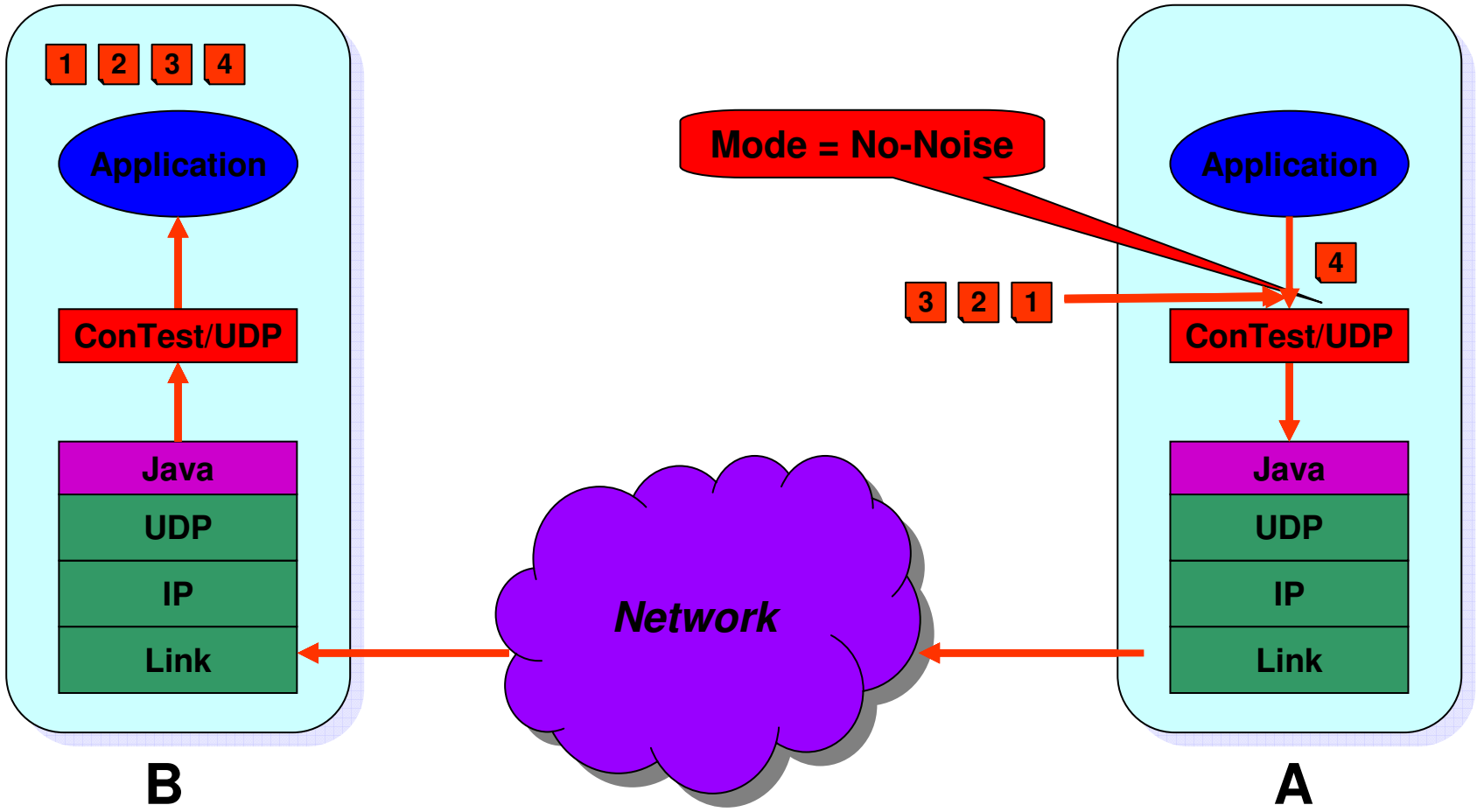
ConTest Noise Heuristics

- Noise Type
  - yields, sleeps, synchYields, random
- Halt One Thread
  - Stop a thread until no other is willing to run
- Tamper with timeouts
- Shared Variables Noise
  - Concentrate on all or one variable
  - Collect shared variables dynamically
- Option to start late
  - At certain class, method
- UDP Noise

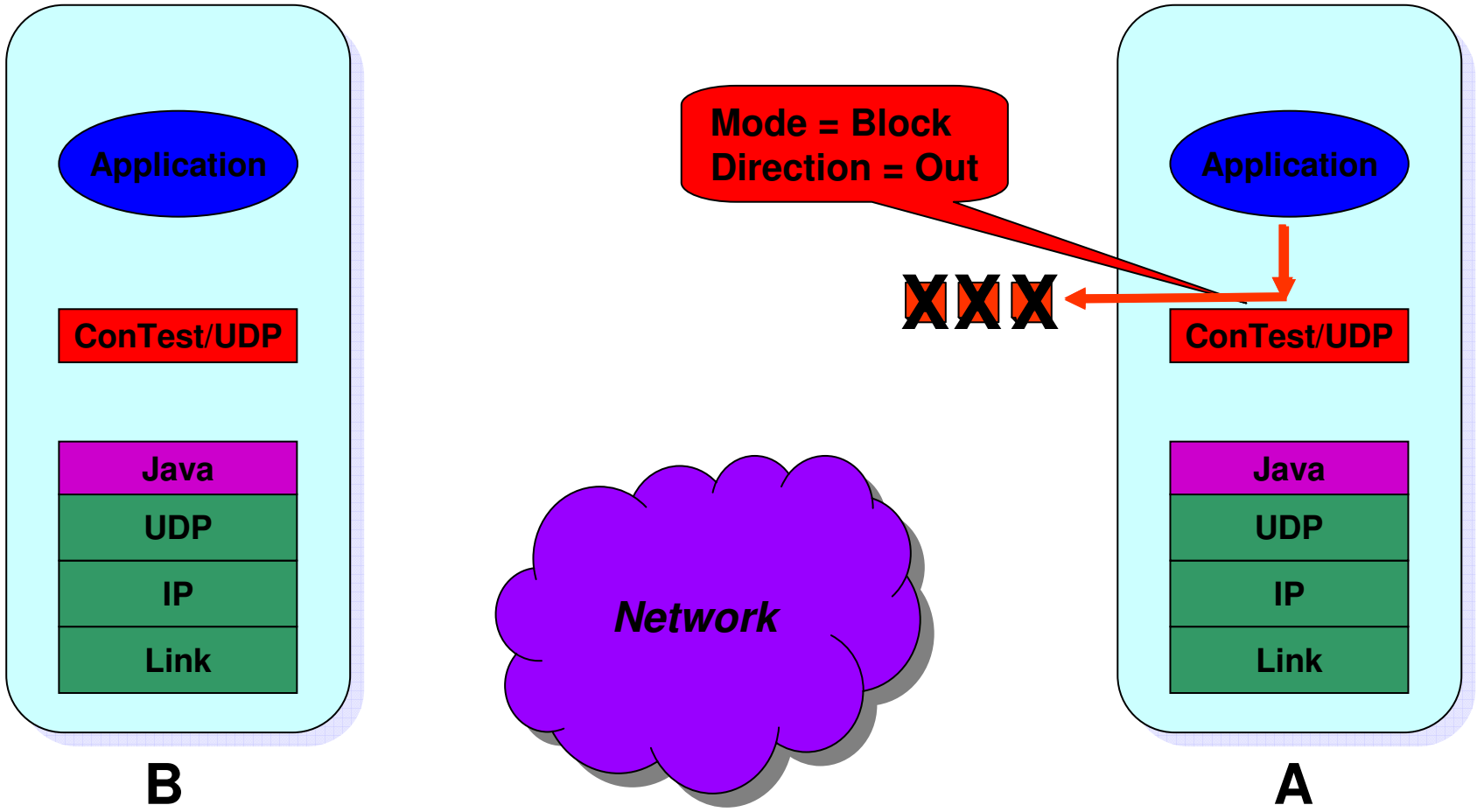
Delaying messages with ConTest/UDP



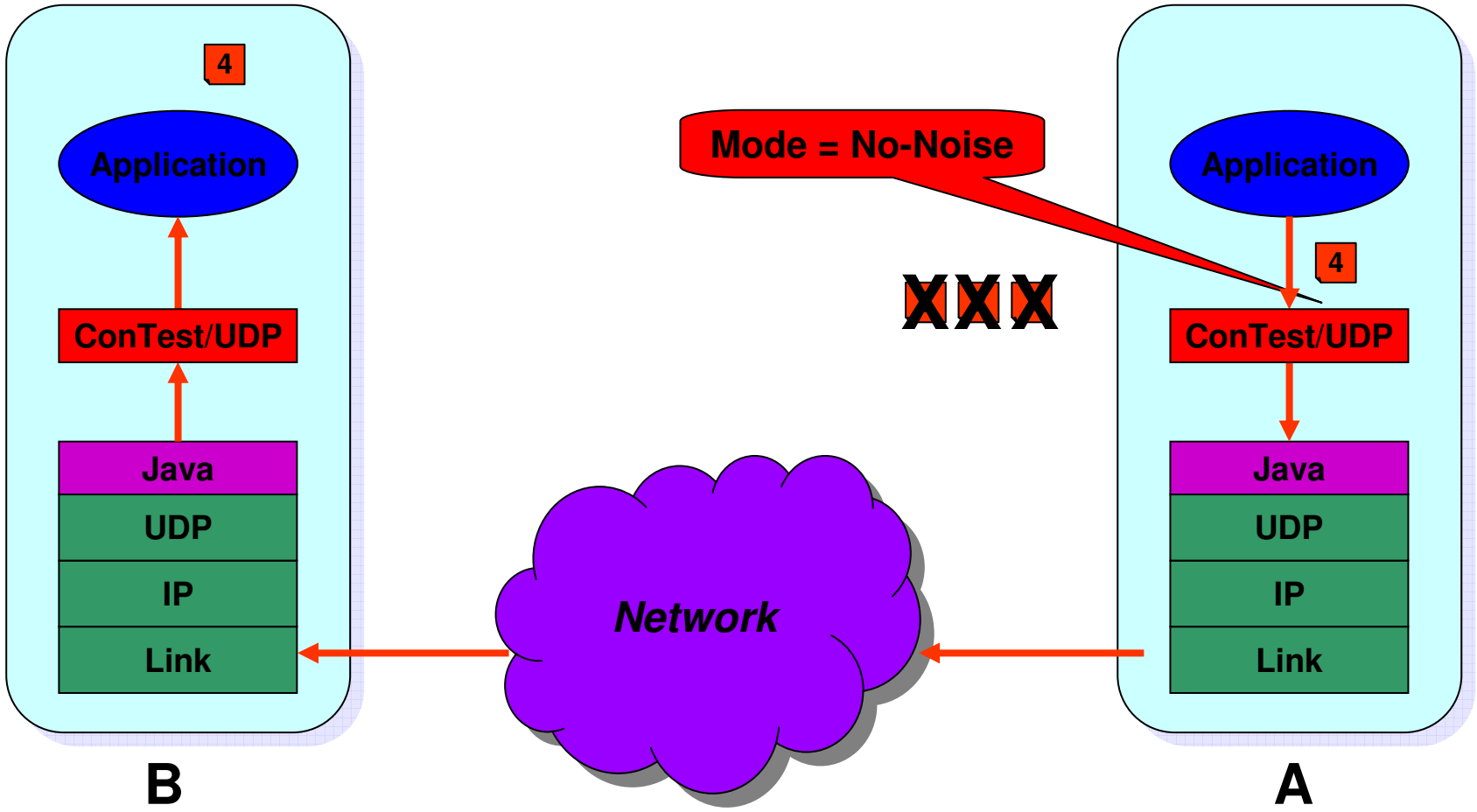
Delaying messages with ConTest/UDP



Losing messages with ConTest/UDP



Losing messages with ConTest/UDP



ConTest Debug Options

- Deadlocks
  - Location of each thread
  - Cycle of waiting on locks
  - Lock taking trace
- Lock discipline violation detection
  - Advanced options: exhibiting 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 on noise injection
- Talk with ConTest (socket/keyboard) while the application runs

Measuring Concurrent Coverage

- ConTest also measures code coverage
  - Standard coverage models such as basic block coverage, method coverage
- How do you know if the tests are any good from concurrency view?
- Synchronization coverage
  - Make sure that every synchronization primitive was fully exercised
- Shared variable coverage
  - Make sure shared variables were accessed by more than one thread
- More systematic metrics exist but they do not scale
- We will later see applications of synchronization coverage

Instrumentations Issues

- In Java, we had three choices: Source, bytecode, JVM
  - We chose bytecode
  - Can be performed offline as a preliminary stage or dynamically at classload (by adding a flag to the run command)
- In C/C++
  - Where to instrument: source, object, libraries
  - Issues to consider: ease of implementation, portability, capabilities
  - Implement for every concurrency library

Other Porting Issues

- Understanding the primitives
  - Vital not to introduce new errors
  - Example: sleep() in Java
- Lock-based vs. interrupt-based
  - So far, worked on lock-based
- Type of appropriate noise

Unit test goals and outline

- Do a thorough test of **each** synchronization protocol.
- Use extraction and mock objects to **isolate** the protocols.
- Use **interleaving review** to guide writing tests.
- Test all corners, cover the **scenario space**.
  - 100% basic block coverage
  - 100% synchronization coverage
- Automate, and run tests all night.
- Verify lock discipline.

Unit Testing Concurrent Code with ConTest

1. Remove non-relevant code (optional where applicable)
2. Create a number of tests
3. Instrument the code with ConTest
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
    3. Genuinely uncoverable, adjust coverage goal, go to 5
6. Done

System Test Goals and Outline

- Program is tested in a “natural” setting (or an approximation to natural setting, e.g. simulation).
- Synchronization protocols are tested indirectly.
- Test as much as possible, according to available resources, with no claim to completeness.
- Code coverage is used to find big “holes”: areas of the code that received little attention in the test.

System Test Goals and Outline (Continued)

- The big question: did the test stress all the synchronization protocols?
  - Not obvious, since the tester usually does not know the protocols.
  - Synchronization coverage helps finding the answer.

Function and System Test with ConTest

- Instrument only the classes containing the protocols.
  - The more classes are instrumented, the more noise ConTest makes.
  - Noise costs runtime: more noise, less tests.
  - Instrumenting only concurrent code will focus the noise where it is most useful.
- After running the tests:
  1. Verify that no bug was found
  2. Check lock discipline
  3. Check coverage

Reviews of Concurrent Code:
The Interleaving Review Technique (IRT)

Code Inspection Methodologies and Tools

- Formalized inspection process was first developed at IBM in the 1970s (Fagan)
- Principal objective is defect detection
- Many methodologies have been suggested
  - Active research area
  - Static and dynamic analysis can be used to support the inspection process
  - Analyze the program with respect to known checklists or bug patterns
- To set the scene for IRT, we will briefly discuss the desk checking methodology

Code Review: Pros and Cons

- Pros
  - Great ROI
    - A major defect every three hours and a defect every half hour
  - Finds bugs early
    - Between 60% and 90% of the bugs can be found in reviews
  - Saves time and reduces cost
    - Finding the fault instead of the failure
  - Builds effective teams
  - The observer effect
  - Educates
  - Increases tester-programmer communication
  - Same process can be effectively applied to all project artifacts

- Cons
  - Skepticism – feeling that something so simple and informal can't be useful
  - Boring
  - Author protection
  - Misconception of project priorities
    - Project is "front loaded"

Comparing Code Reviews and Testing

- Code reviews have lower debugging cost
  - When an error is found, the precise nature and/or location of the error is usually found
  - Testing produces a symptom of the error
    - Further analysis is required to locate the bug
    - The analysis becomes harder as you go from unit test to functional test to system test
- Code reviews expose a batch of errors that can be corrected in one shot
  - Testing usually detects errors “one by one”
- Code review and testing are effective in revealing different types of bugs; but there are many bugs that can be found effectively by both processes
  - Testing and code review are complementary processes
  - For example, an unreleased resource along an error path is found more effectively in code review
  - However, code review has pitfalls that testing does not have
    - The reviewer adopts the wrong assumptions of an implementer
    - The reviewer misses interaction bugs due to too many details
- Automatic regression suites create a safety net that is easy to apply

Desk Checking (Walkthroughs)

- An extremely effective code review technique used for early detection of sequential program errors
  - Desk checking means manual execution of the program
    - Writing the tests first.
    - The system behavior is reviewed
- Introduced in 1974 by C.A.R. Hoare in his structured programming course
  - “The first principle of error detection is that the sooner an error is detected the less trouble it will cause”
  - Back then, most written programs were sequential

A Toy Example of Desk Checking

- Program definition – sum up the positive integers that are smaller than j
- Program segment

```
int sum = -1;
int j = 0;
read(j);  //The user inputs j
for(int i = 0;i < j; i++)
   sum = sum + i;
```

- Tracing of test data j = 1
- Bug found – sum = -1 at the end

control	sum	i	j
	-1		0
read(j)			
	-1		1
i=0			
	-1	0	1
(i<j) is true			
sum-sum+i			
	-1	0	1
i++			
	-1	1	1
i<j is false			

Walkthrough Roles

- **Tester –** comes to the meeting armed with a small set of paper test cases or scenarios
- **Program counter –** plays computer
  - This is usually the program owner
- **Inspector** – finds errors, omissions, inconsistencies, or identifies broader issues
- **Moderator** – leads the inspection team and is responsible for ensuring good inspection
- **Scribe –** records the results of the inspection meeting
- **Owner** – responsible for fixing defects discovered during the review process

Walkthrough Mind Set

- Similar to testing mind set
  - Review test cases for invalid and unexpected, as well as valid and expected, input conditions
  - Do not plan tests under the tacit assumption that no errors will be found
  - The probability of the existence of more errors in a section is proportional to the number of errors already found there
  - A good test case is one that has a high probability of detecting an as-yet undiscovered bug
  - Avoid throwing away test cases unless the program is truly a throw-away program
- Inspect the intermediate states of a program as it is reviewed
  - Determine if the program does what it is not supposed to do
- Sometimes test cases engender discussion about program logic and assumptions

Code Base Coverage

- While manually executing the code, note:
  - Statements that were never executed
  - Decisions that were never taken
  - Values that should impact the flow but never did
  - Possible interface results that were never assumed
  - Possible environment states that were never assumed
- Such things can motivate additional desk checking
- Later, during testing, use code based coverage (supported by ConTest) to check the same things

Desk Checking is for Sequential Code

- If a concurrent program is manually executed
  - It is not clear which process to advance at each stage
- Test selection should take into account a new space
  - The space of possible interleavings
- Desk checking is applied to a single artifact whereas some concurrent bugs may only be discovered by examining several artifacts simultaneously

The Interleaving Review Technique

- Derivation of the desk checking (walkthrough) technique
  - Design review and code review oriented towards
    - Concurrency
    - Fault tolerance
  - In addition to traditional review
  - Helps in test plan design
- When combined with ConTest
  - Provides a lightweight concurrency oriented code verification technique

Adoption

- Used successfully for middleware projects in IBM
  - Found additional problems in reviewed code each time it was used
  - Minimal additional effort
- Adopted for use in all new code developed
  - Developers see the benefit
  - Short learning curve
- Some statistics gathered

IRT Statistics from Experiments with Already-reviewed Code

Project type	Overall time spent (hours for all developers)	Number of bugs found	Comments
Cluster device drivers	2	2	2 developers
Cluster device drivers	10	15	5 developers. Was done after 6 hours of regular reviews
Cluster device drivers	12	4	2 developers
Cluster device drivers	1	1	2 developers
Cluster device drivers	2	1	2 developers
Cluster device drivers	3	2	3 developers
Group communication	10	1 design, 2 bugs, 17 code modifications	2 developers
Group communication	5	2 bugs, 4 code modifications	2 developers

The Problems IRT (Interleaving Review Technique) Addresses

- When attempting to review or test the system behavior of a concurrent/distributed and fault tolerant system, several problems arise
  - Non-determinism
    - Given that the program is in some state, the next program state depends on which process executes next
      - As a result, it is not clear how to proceed with the review process
- The state of the concurrent or distributed program is determined by the state of all its processes and their interrelated temporal dependencies
  - 3 processes with 10 states have 1000 possible states to review
- Tests are much more expensive
  - Require the interaction of many machines and failures in predetermined sequences
    - Not necessarily repeatable

IRT Consists of

- The use of the Cartesian product technique to select interleavings and states to review (FoCuS)
- Definition of review roles and guidelines to carrying out the roles
  - **Program counter** – needs to thoroughly understand the system so he can determine the control flow
    - If several components are exercised in the scenario under review, the program counter role may be fulfilled by several component owners
  - **Devil's advocate** – experienced in concurrent and fault tolerance systems. His role is to make choices regarding the timing of events and failures
    - To maximize the probability that a bug is found
    - IRT provides guidelines for making these choices
  - **Stenographer** – experienced in representation techniques (use cases, sequential diagrams, time diagrams, etc) and able to strike a trade-off between accuracy and readability

Why Let a Different Process Advance After a Lock is Obtained?

- The devil's advocate decides that another process/thread advances right after, or before, a synchronization operation is performed
  - He also makes sure that locks are waited on
- Most of the synchronization primitives require that all processes accessing the shared resource follow the protocol
  - Thus, obtaining the lock does not guarantee protection if other processes are not attempting to obtain the same lock
- This choice of method significantly decreases the number of interleaving to consider for review

A Toy IRT Example – Who is the King?

- Code segment executed by several processes with the objective of choosing a leader processor

```
boolean chosen= false;  // global variable used for process coordination
boolean ImAKing = false; // local – indicates the current process status
if (chosen == false) {
   lock();
      chosen = true;
      ImAKing = true;
   unlock();
}
```

An IRT Example – Who is the King?

Process one	Process two	Chosen	Process one ImAking	Process two ImAking
Program Counter - Start executing		false	false	false
If (chosen==false) is true		false	false	false
Devil advocate- Advance second				
	If (chosen==false) is true	false	false	false
	lock()	false	false	false
	chosen=true	true	false	false

```
boolean chosen= false
boolean ImAKing = false
if (chosen == false) {
 lock();
   chosen = true;
   ImAKing = true;
 unlock();
}
```

An IRT Example – Who is the King?

Process one	Process two	Chosen	Process one ImAKing	Process two ImAKing
	ImAKing =true	true	false	true
	Unlock()			
Devil's advocate- advance first				
lock()		true	false	true
chosen = true		true	false	true
ImAKing = true		true	true	true
unlock()		true	true	true

```
  boolean chosen= false
  boolean ImAKing = false
if (chosen == false) {
  lock();
    chosen = true;
    ImAKing = true;
  unlock();
}
```

Some Guidelines for the Devil's Advocate

- Increase contention on shared resources
- Delay locks so that locks are obtained in different orders
- While in critical section
  - Force error paths; assume that potentially blocked operations are blocked and cause signals and interrupts to occur
- Cover all possible scenarios of waiting on event
  - Event notification is sent
    - Before and after the event is waited on
  - If waiting on event is not atomic, event notification is sent after the event is checked and before it is waited on
- Break assumptions that depend on hardware and scheduler
  - Assume that delays are not long enough
  - Assume that changes are not visible due to the memory model
- Based on concurrent bug pattern paper (PADTAD2003)

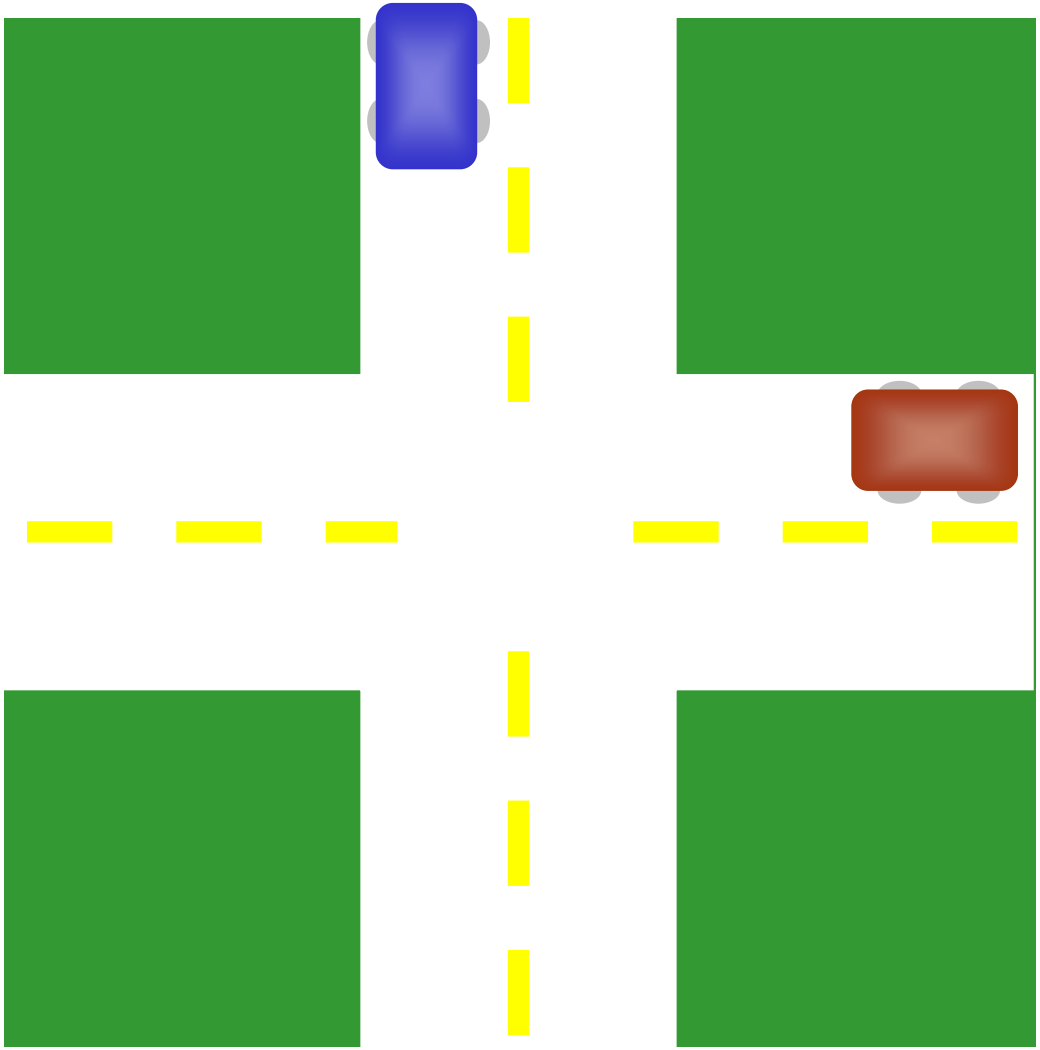
SHADOWS:
Self-Healing of Concurrent Systems

The Concurrency Healing Process

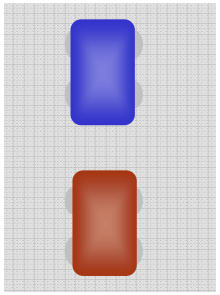
1. Testing
2. Debugging
3. Fixing – or suggesting fixes
4. Verifying

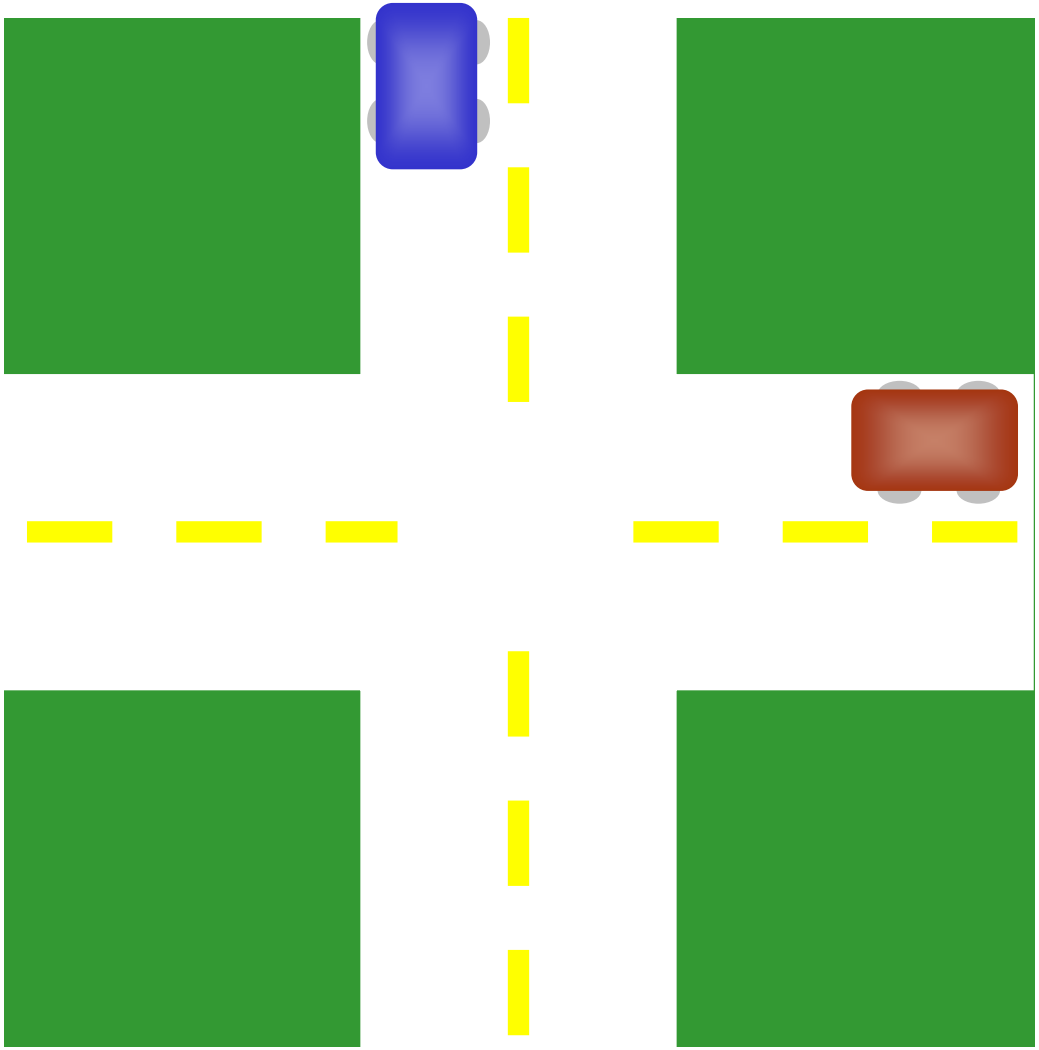
- All done automatically

What is Concurrent Healing About

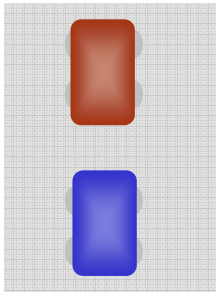


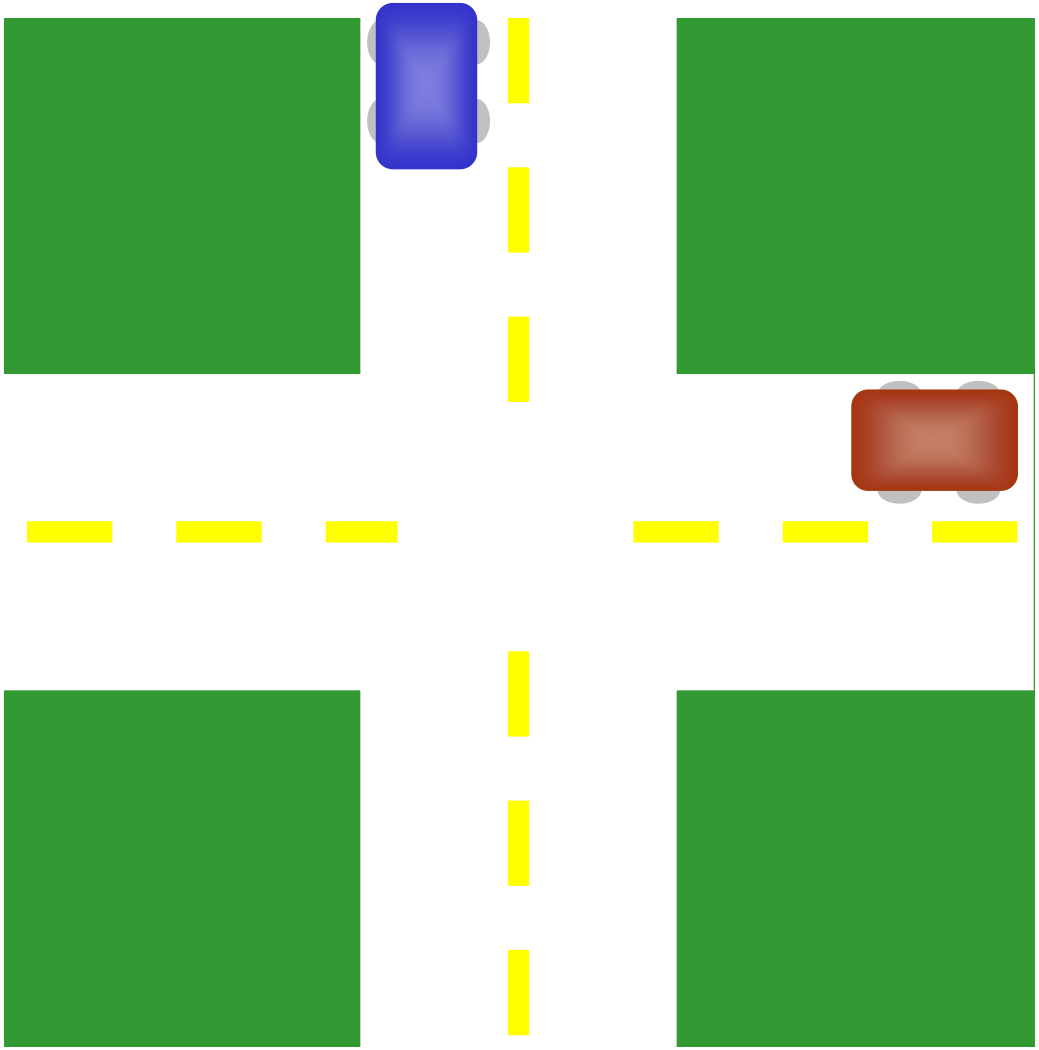
What is Concurrent Healing About



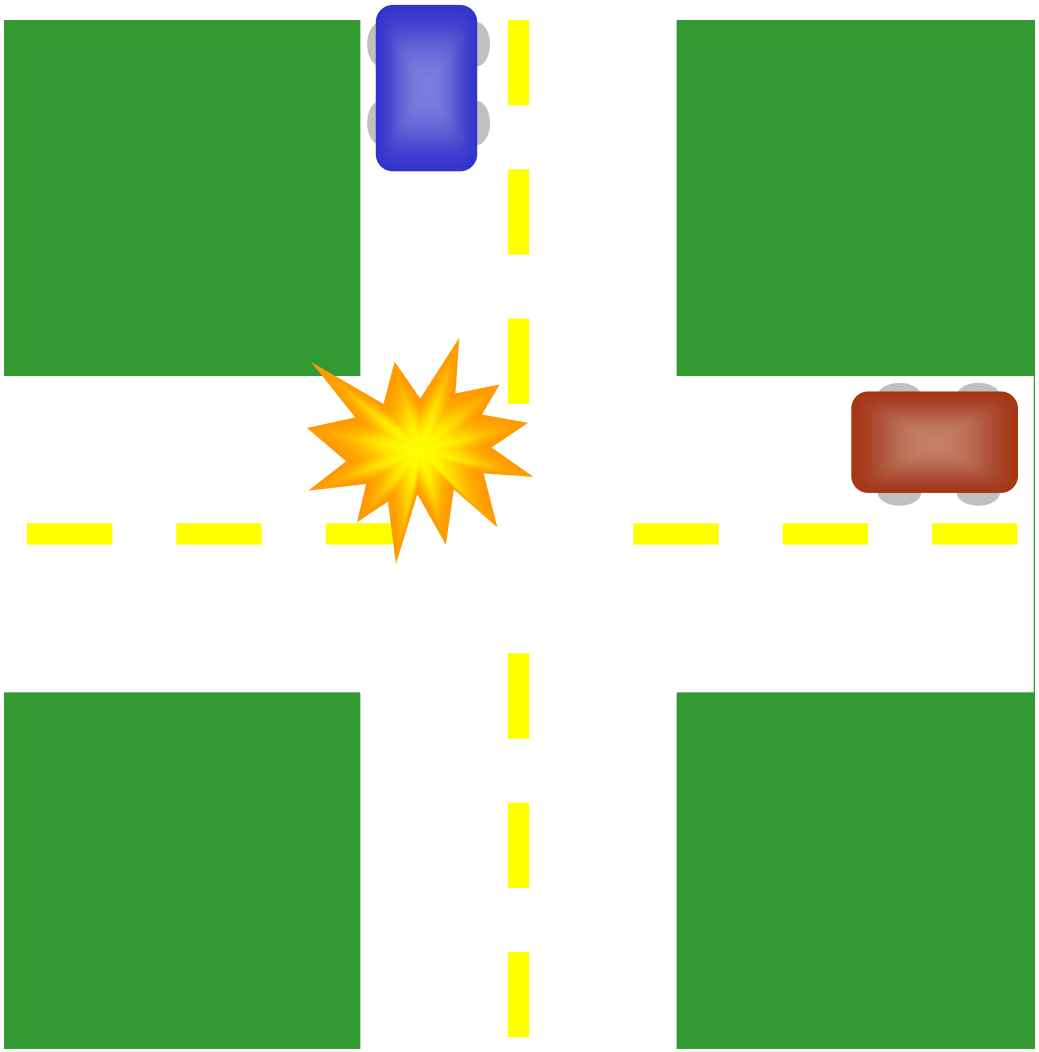


What is Concurrent Healing About

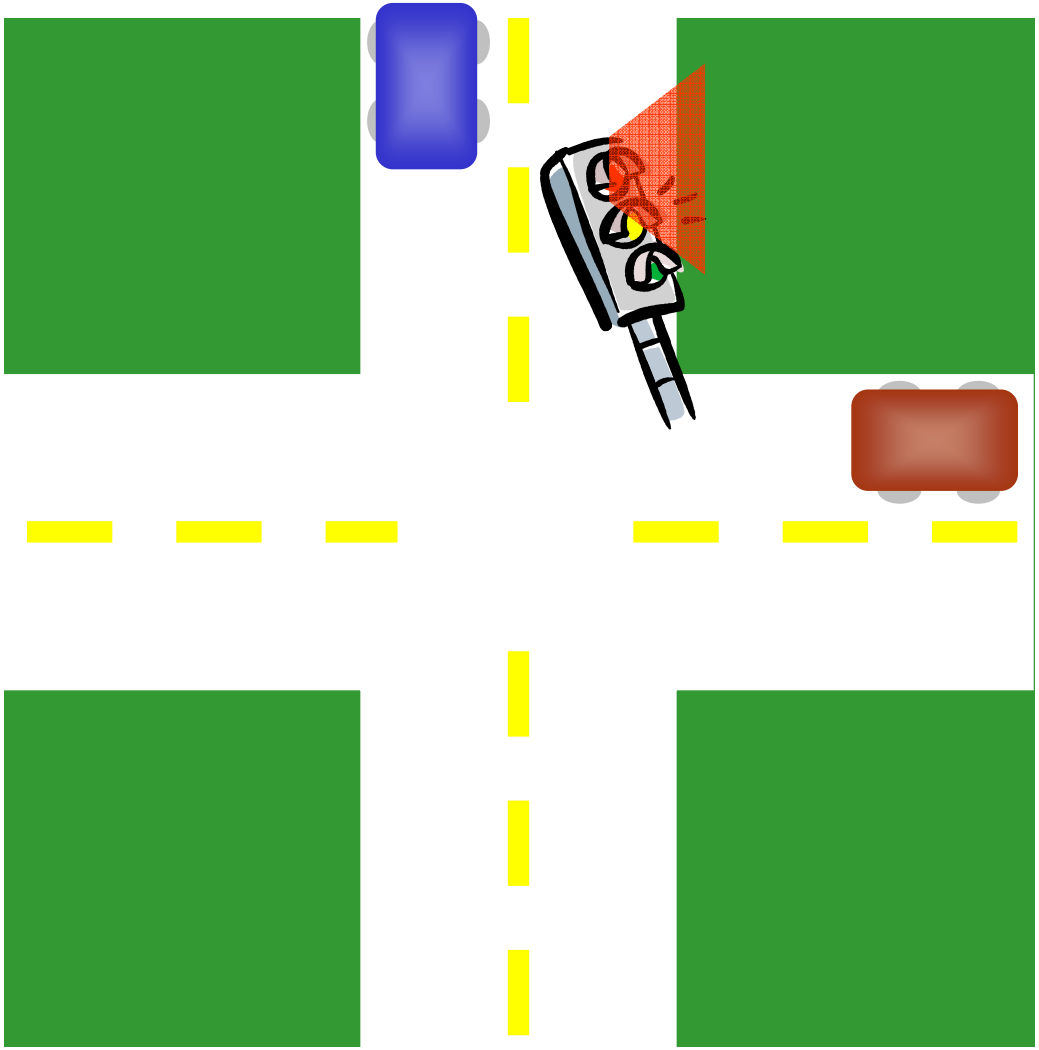




What is Concurrent Healing About



What is Concurrent Healing About



Listener Architecture

- Allow other parties to write extensions to ConTest
- Use ConTest for:
  - Instrumentation
    - Developers write extension code to be executed when target program performs certain events
    - The architecture takes care of invoking the extension code
  - User base
    - Existing users of ConTest can easily validate the extension
    - Can easily use extensions - no need to install new tools

Listener Interfaces

```
class MyExtension implements BeforeMonitorEnterListener,
      AfterMonitorEnterListener, ... {
   public void beforeMonitorEnterEvent(...) {
      ...
   }
   public void afterMonitorEnterEvent(...) {
      ...
   }
   ...
}
```

Listener Interfaces

- These methods will be called whenever the target program synchronizes on some objects.
- Methods get access to context information:
  - which object is synchronized on
  - where in the code
- Other listenable events: monitor release, member read & write, thread start & end, various methods
- Extensions are registered with ConTest by placing an XML file in a predetermined directory
- Extensions can also be implemented as extension plug-ins to ConTest plug-in for Eclipse

Listener Interfaces

- Architecture also provides useful utilities
  - API for Quick and replayable random noise
  - Replay of parameters
  - Deferring activity to a late point in execution
  - Query lock status and threads location
  - Safe treatment of target program objects
    - Avoiding some caveats
  - Uniform screen output
  - Uniform file output (sensitive to test run scope)

Possible Uses

- All classical concurrent testing and debugging tools
  - Scheduling interference
  - Race detection
  - Atomicity checking
  - Lock discipline
  - Code coverage
  - Synchronization coverage
  - Tracing and Debug information

Automatic Debugging Approaches

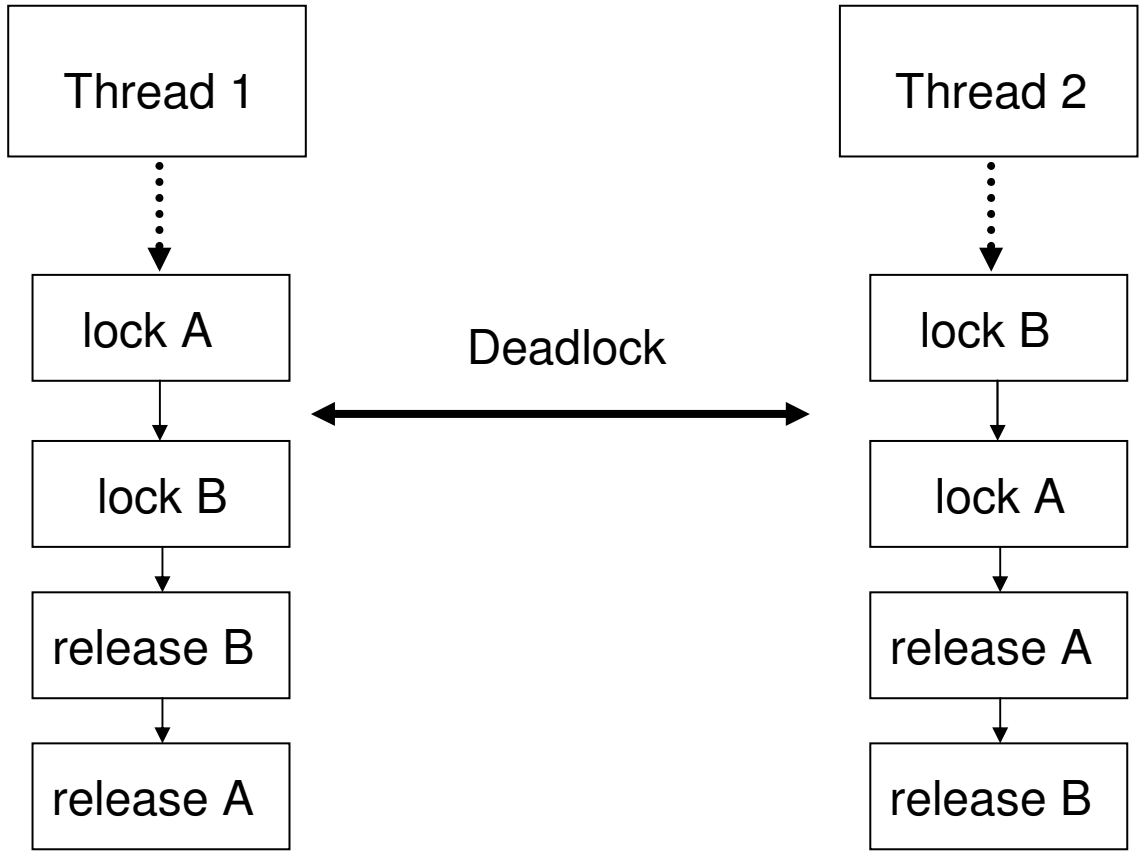
- Online
  - Identify specific bug patterns at run-time
  - Many detection algorithms exist
    - E.g., for races, lock discipline, atomicity violation

- Multiple executions
  - Look for a few instrumentations that reveal the bug
  - Are the found instrumentations useful in analyzing the bug?

Online detection algorithms implemented on top of ConTest

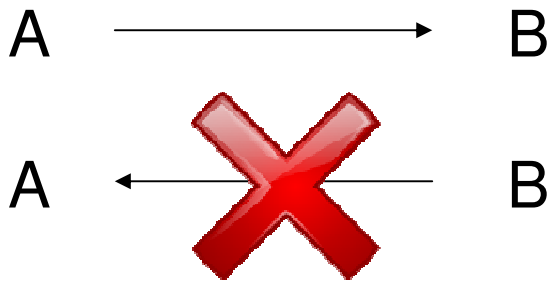
- Deadlock exhibiting
- Deadlock Healing
- Race Detection (Eraser)
- Atomicity violation detection and healing
- Synchronization coverage forcing

Classic Deadlock Situation



Lock Discipline

- Nested locks should be taken in a predefined global order



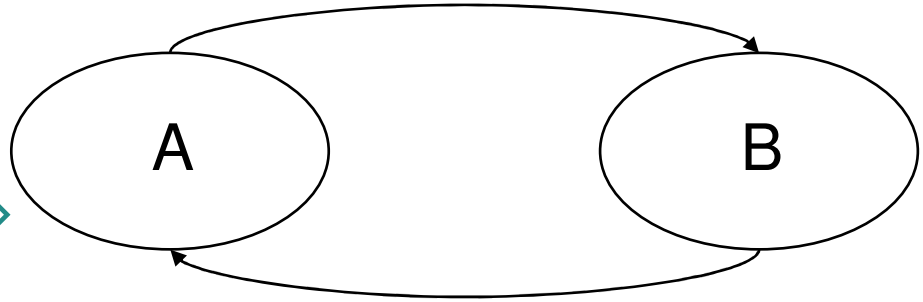
- If lock discipline is maintained, no such deadlocks exist

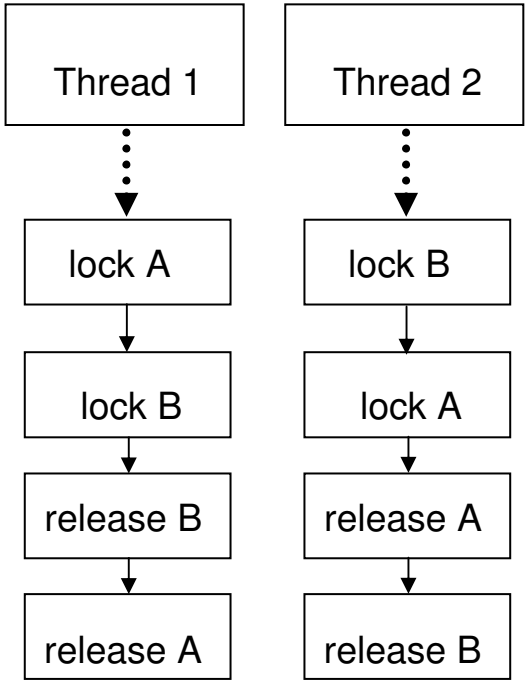
Gate Locks

- A lock g is a gate lock for a set of locks S if g is always held while locks from S are taken nestedly
- If S has a gate lock, no deadlocks resulting from lock discipline violation among S locks can occur

Lock Discipline Violation Detection

- Build a directed graph of lock acquisitions





- Unguarded cycles are potential deadlocks
  - No common gate lock is held while the locks in the cycle are held
- Even when not real deadlocks, prone to become ones as the code evolves

Cross-Run lock Discipline violation Detection

- Required since different parts of a cycle may be seen in different runs
- Identifying locks across different runs:

1. According to the program location where the lock is taken
2. Two different program locations are the same lock if there is a run where they are associated with the same lock object
3. The set of locks is a transitive closure of the same lock relation

Deadlock Exhibiting

- Determining whether a potential deadlock is real is time consuming
- Noise injection is a scalable method for exhibiting concurrency bugs
- Exhibit deadlocks at testing phase using targeted noise:
  - Identify unguarded strongly connected components (SCCs)
  - Before a thread takes a lock from an SCC while holding another, inject noise
- Implemented as a ConTest listener
- [Havelund et al. 06] use deadlock potential information to exhibit deadlocks by scheduling the threads through a global controller
  - More control but also more intrusive and less scalable

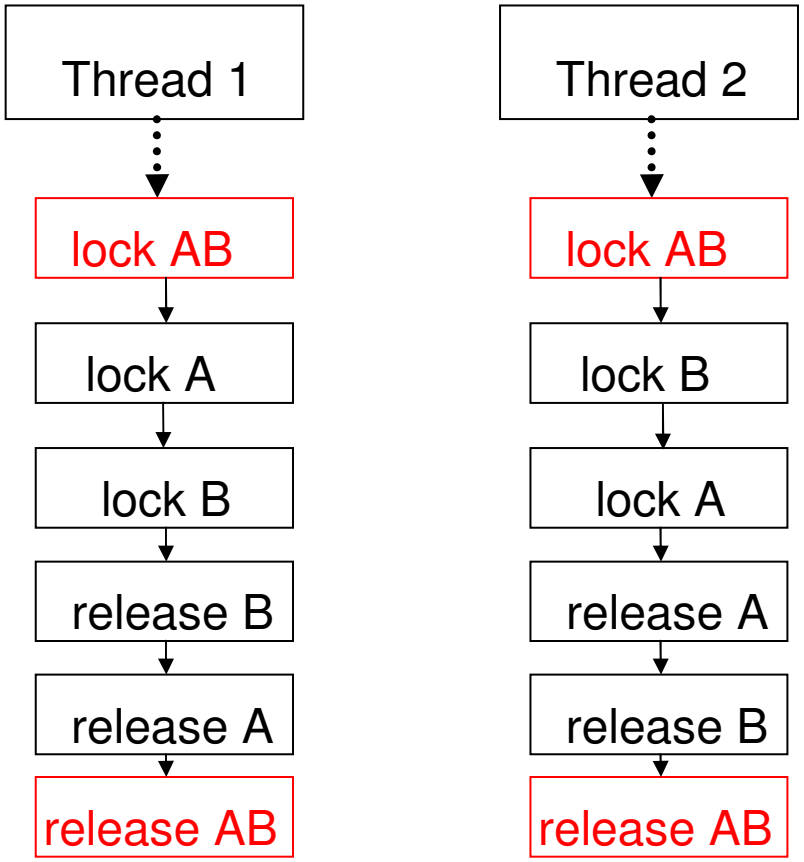
Deadlock Healing

- Why healing?
  - A program is running in the field and needs to run correctly without waiting for code fixes
- Deadlock avoidance and prevention:
  - Declare required resources up-front
  - Kill a process involved in a deadlock
  - Both not applicable to deadlocks in multithreaded programs

Our Deadlock Healing

For each unguarded SCC, add a wrapper (gate) lock

- Taken before any lock in the SCC is taken
- Released after all locks in the SCC are released



Our Deadlock Healing: implementation and pitfalls

- Very easy to implement
  - Does not require run-time monitoring
  - Could be implemented by code change
  - However some complications exist…

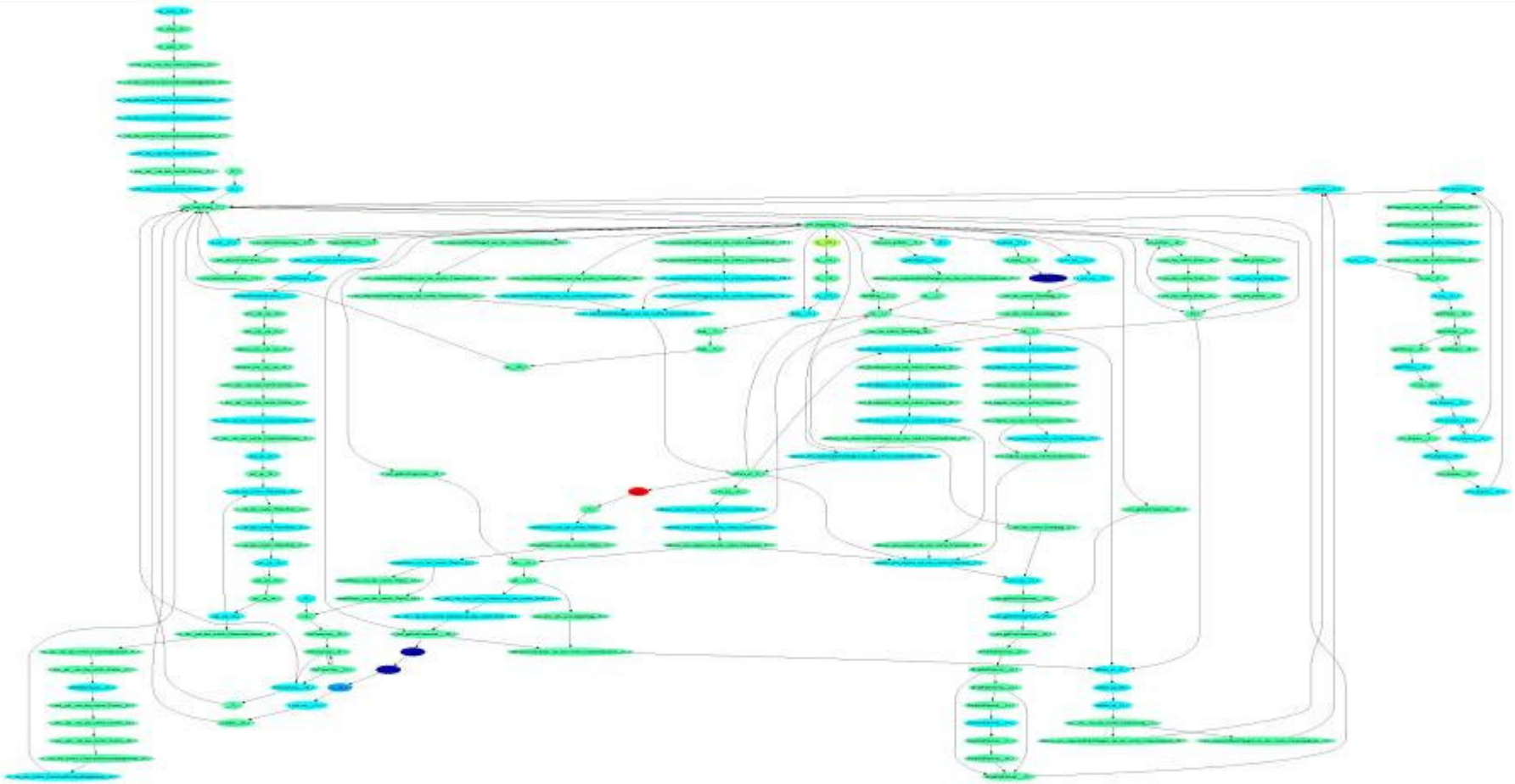
- Incomplete input (missing information about SCCs) can lead to new deadlocks involving a wrapper lock
- Wait synchronization primitive can cause problems..
- We handle these complications in our solution
- Implemented as a ConTest listener

Automatic Debugging – Multiple Executions

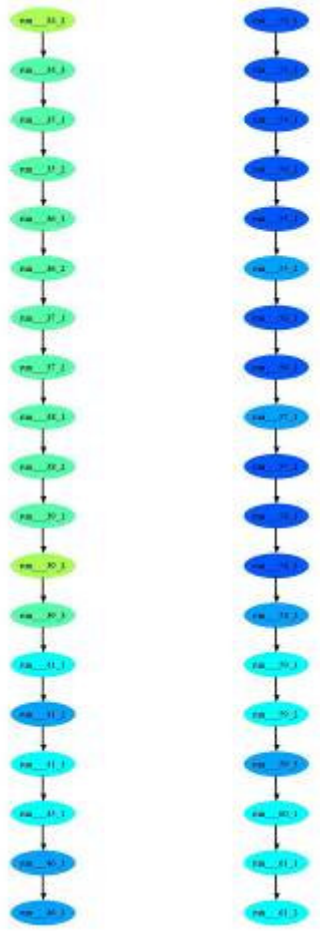
- Input: a test that sometimes fails and sometimes pass (for the same input)
- Goal: automatically pinpoint the locations in the code that are related to the root cause of the bug
- Means: feature selection
  - Think of each instrumentation point (program location) as a feature
  - Execute the test many times with noise injection in different subsets of points
  - Each instrumentation point gets a score

    $P(i) = P(success|X_i)/P(!success|X_i)$
  - The points with the highest score are related to location of failure
  - Can handle the non-monotonic nature of the problem

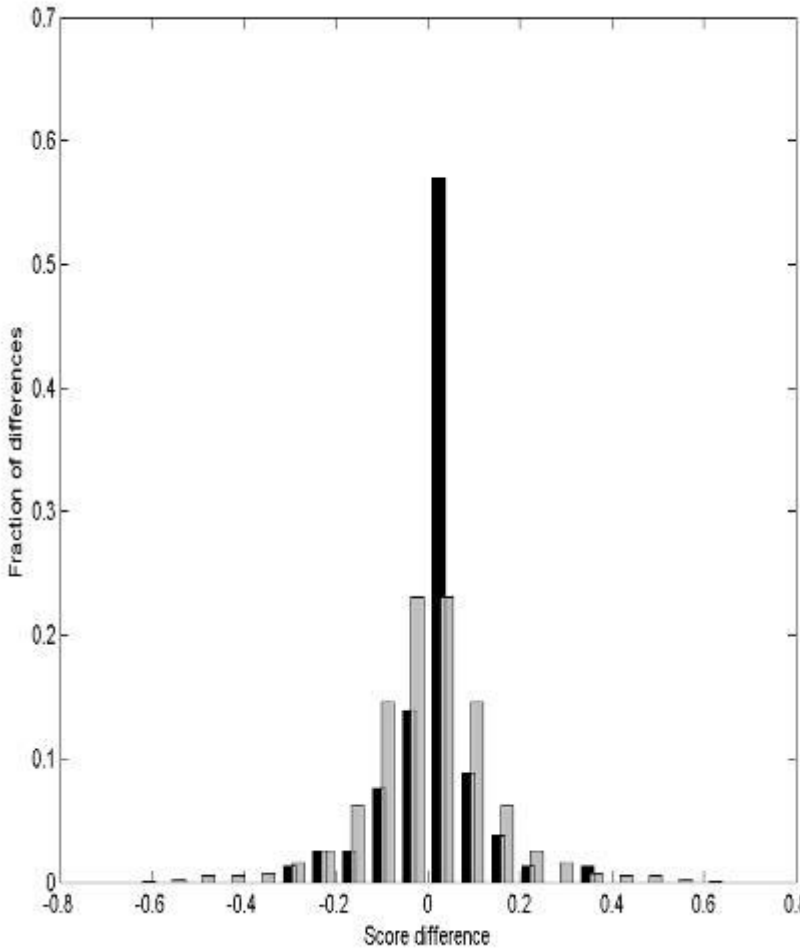
First Program We Tried It On (Crawler in WebSphere) Seems to work

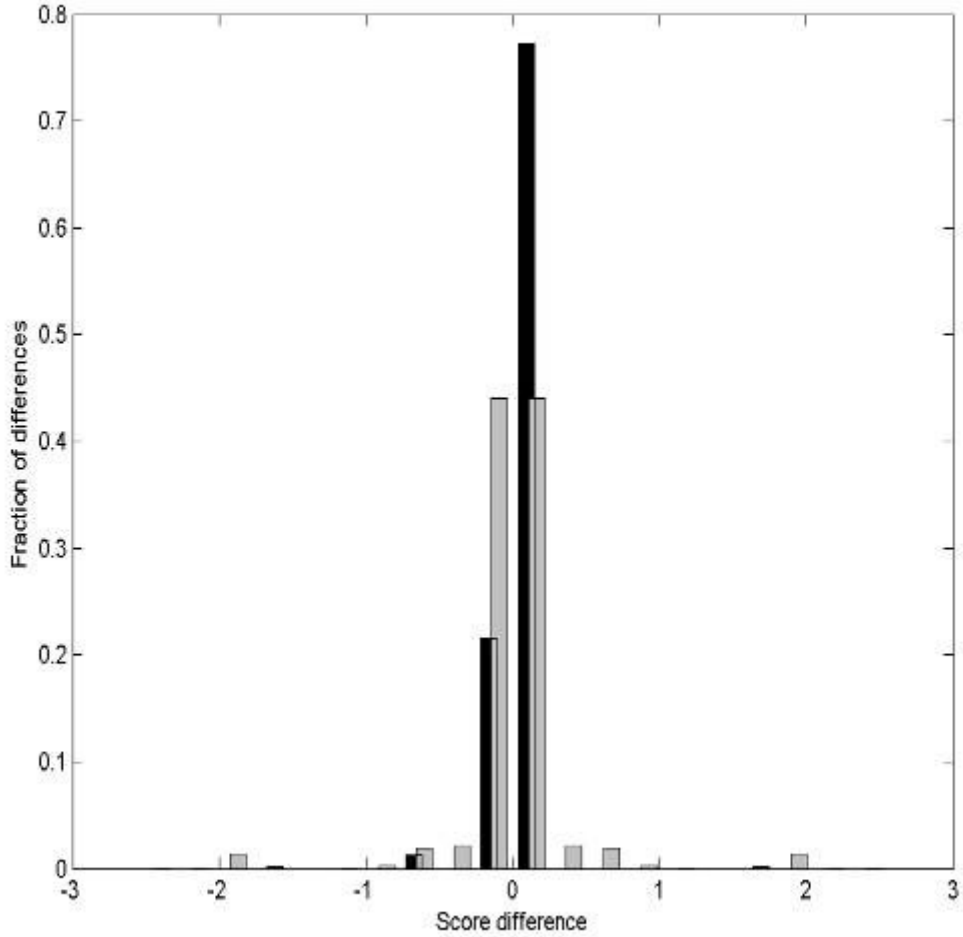


This does not pinpoint when the problem is easy

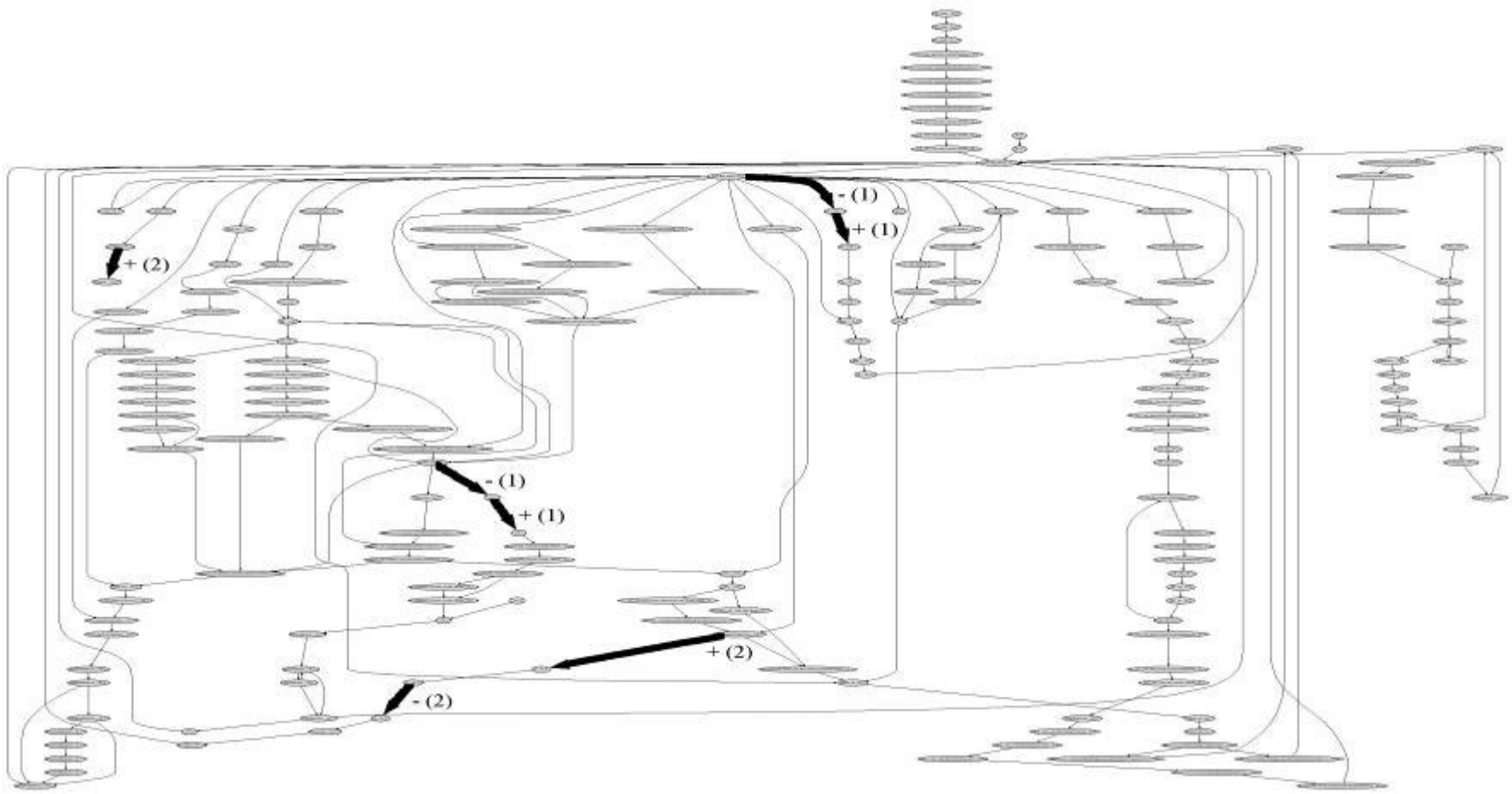


Distribution of Differences Between Pairs of Program Locations

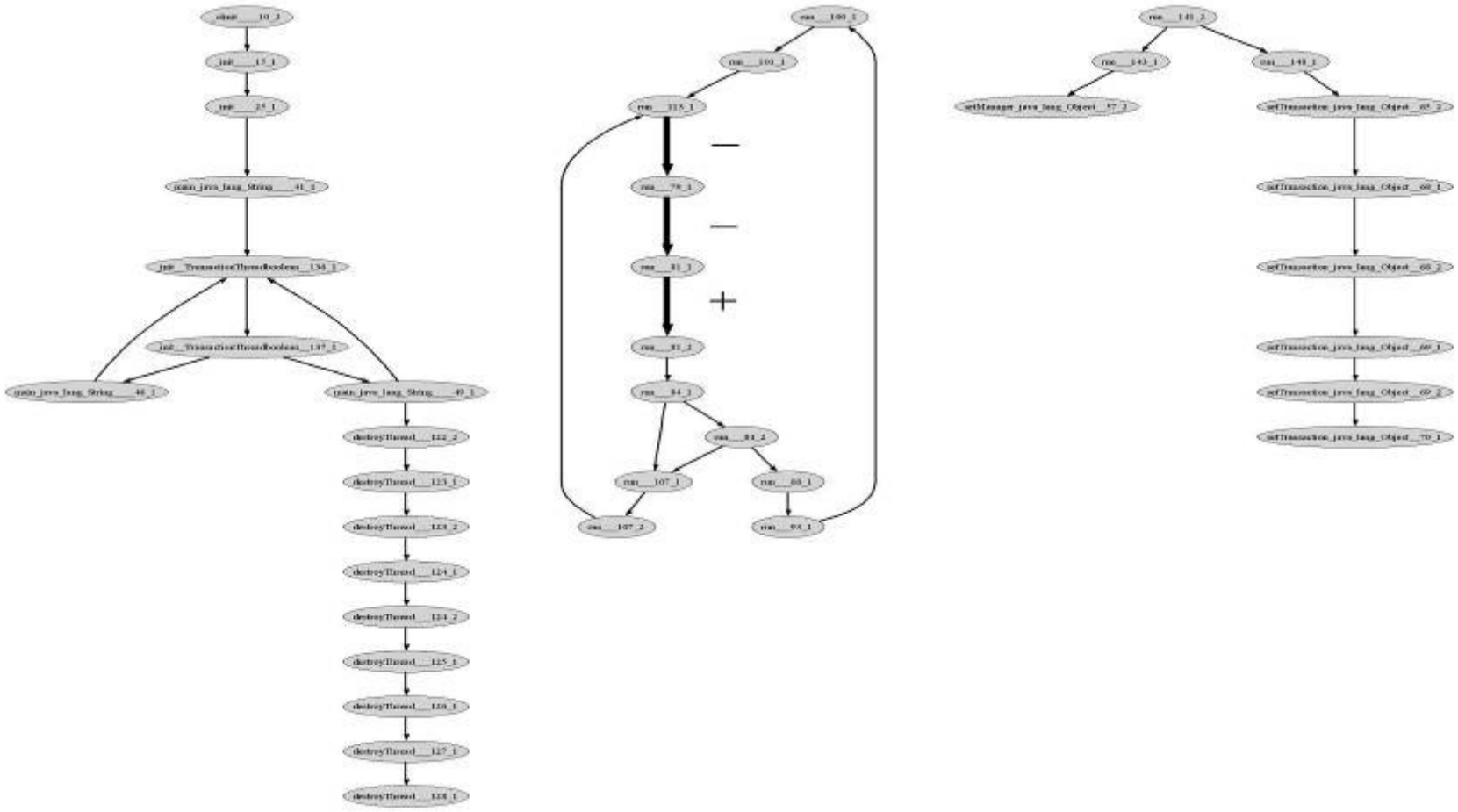




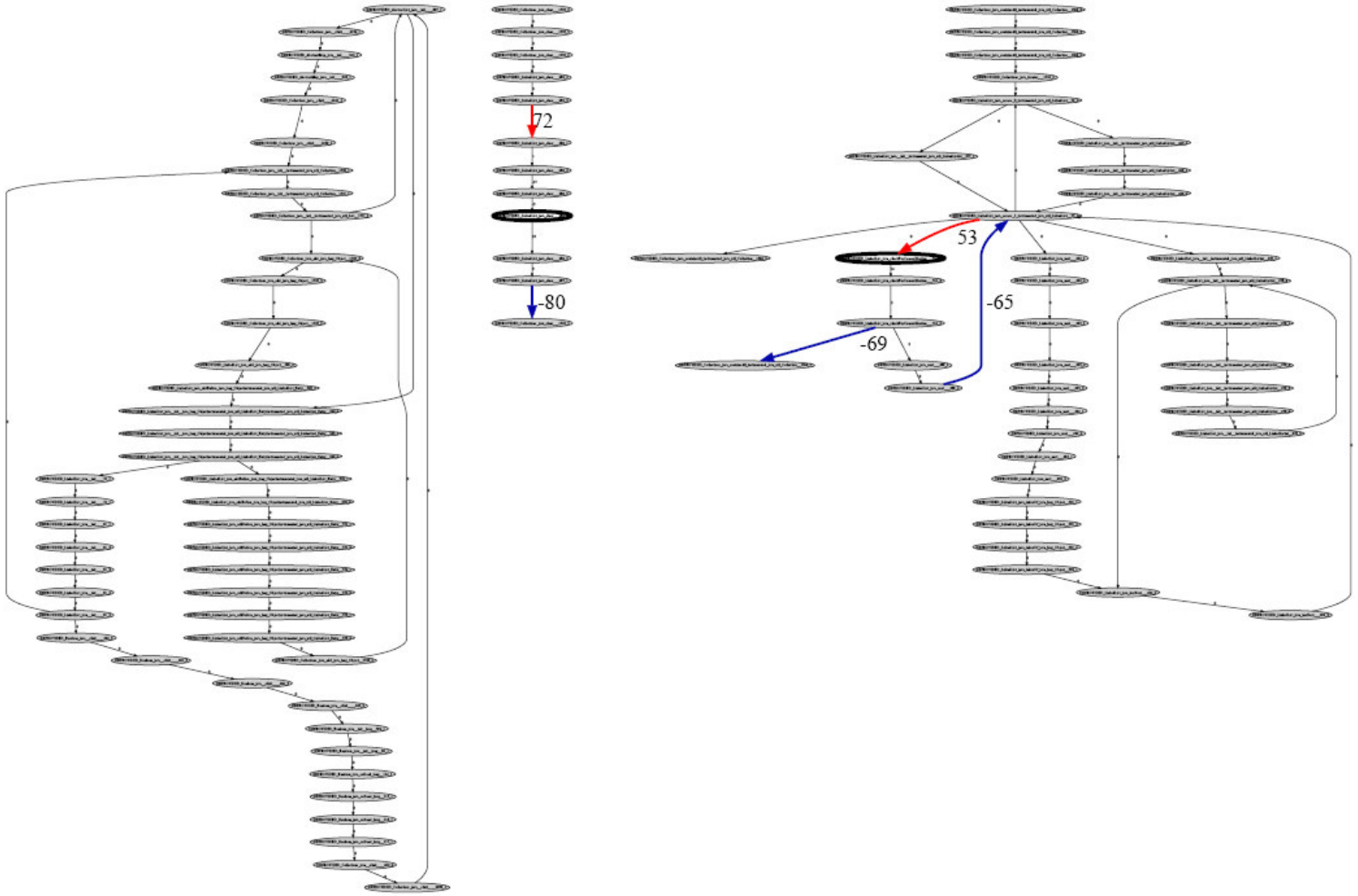
The Derivative along the CFG of the First Large Program



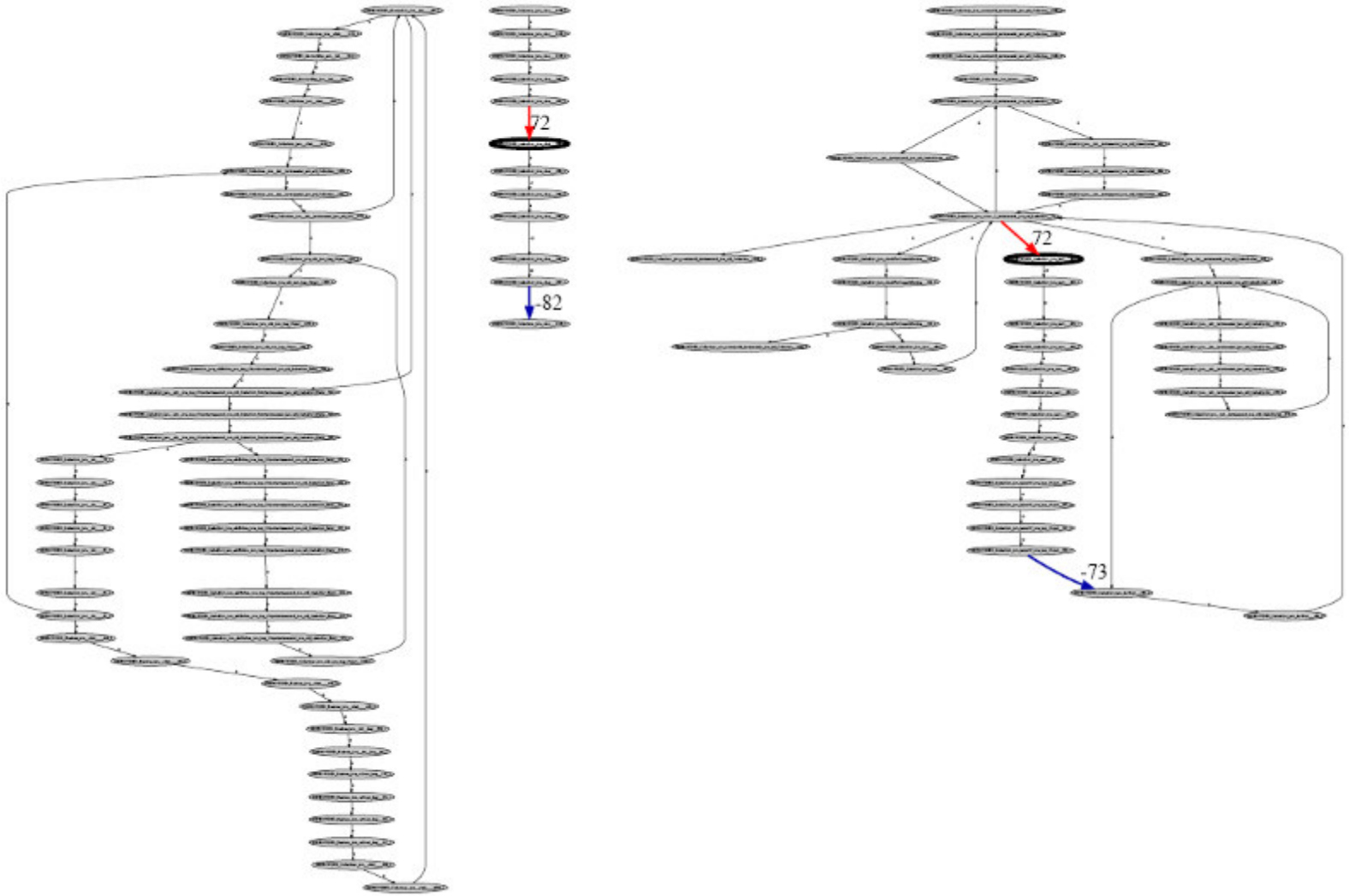
The Derivative along the CFG of the Second Large Program



Java 1.4 Collection Library - Element exception bug



Java 1.4 Collection Library - Infinite loop bug



Automatic Debugging - Future Work

- Use root cause analysis for healing
  - The dual problem: find the best points that conceal the bug
- Better search on instrumentation
  - More efficient – incorporate coverage information
  - More accurate – search for pairs of points
- More experience with industrial programs
  - Already found a root cause of a race in a SHADOWS validator program with ~300 classes and more than 200000 instrumentation points
  - Improve performance further to make technique industry-ready

Summary

- Building concurrent software is difficult – do it only when necessary
- Ensure these components are in place
  - Education
    - Teach developers how to design, review, and unit test
    - Teach testers how to test it
  - Tooling
    - Use dedicated tools for concurrency testing
    - Support review with tools
    - Use special unit testing automation and coverage tool