Active Testing for Race Detection Homework

Introduction

Active testing has recently been introduced to effectively test concurrent programs. Active testing works in two phases. It first uses imprecise off-the-shelf static or dynamic program analyses to identify potential concurrency bugs, such as data races, deadlocks, and atomicity violations. In the second phase, active testing uses the reports from these imprecise analyses to explicitly control the underlying scheduler of the concurrent program to accurately and quickly discover real concurrency bugs, if any, with very high probability and little overhead. In this homework, you will be implementing the RaceFuzzer algorithm described in the following paper:

K. Sen, "Race Directed Randomized Testing of Concurrent Programs," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08), 2008, pp. 11-21.

Requirements

Windows or Linux or Mac OS X. You need pre-installed Sun's JDK 1.5 for Windows or Linux, or Apple's latest JDK for Mac OS X. You also need Apache's ANT (http://ant.apache.org/) for building and running your code.

Installation

The infrastructure required to implement RaceFuzzer can be downloaded from http://srl.cs.berkeley.edu/~ksen/calfuzzer.tar.gz. Make sure that java, javac, and ant are in your PATH. Invoke the following commands to install the infrastructure.

```
tar zxvf calfuzzer.tar.gz
cd calfuzzer
ant
```

We have provided a simplified implementation of DeadlockFuzzer described in the following paper.

P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), 2009.

This is a reference implementation to show how the infrastructure works. You can test DeadlockFuzzer by running the following command.

```
ant –f run.xml deadlockfuzzer
```

In this homework, you will run

```
ant –f run.xml hw2
```

and make sure it finds race conditions in the benchmark programs. Take a look at the target "hw2" in run.xml to get acquainted with the benchmark programs on which you will be running your implementation.

Description

Read the papers raced.pdf and hybrid.pdf provided with the infrastructure to understand the algorithms that you are going to implement. Read the paper activetool.pdf to get a high-level description of the infrastructure.

You need to implement the classes

calfuzzer/src/javato/activetesting/HybridAnalysis.java
calfuzzer/src/javato/activetesting/RaceFuzzerAnalysis.java

Open these classes and see the inline comments. HybridAnalysis.java need to implement the Phase I described in the raced.pdf and RaceFuzzerAnalysis.java need to implement Phase 2 described in the same paper.

An implementation of an active testing algorithm, such as RaceFuzzer, AtomFuzzer, or DeadlockFuzzer, works in three steps:

In Step 1, the code under test is instrumented using the SOOT Compiler framework. The target "instr" in run.xml shows the actions taken by the instrumentor. The target "test_race1" in run.xml shows how to invoke the target "instr". The instrumentor inserts various static method calls inside the bytecode under test. The set of methods that are inserted by the instrumentor is the set of all static methods starting with "my" in the file calfuzzer/src/javato/activetesting/analysis/ObserverForActiveTesting.java. The

infrastructure provides an implementation of the instrumentor and you do not need to re-implement it.

In Step 2, the instrumented code is run once with an imprecise dynamic analysis, such as Hybrid Race Detector or iGoodlock. The target "analysis-once" in run.xml shows the action taken by an imprecise analysis. The following

```
<antcall target="analysis-once">
     <param name="javato.activetesting.analysis.class"
     value="javato.activetesting.HybridAnalysis"/>
</antcall>
```

shows how to invoke an imprecise analysis (Hybrid Race Detector in this case) by specifying the Java property "javato.activetesting.analysis.class". An analysis class, such as HybridAnalysis needs to extend the abstract class javato.activetesting.analysis.AnalysisImpl (which implements the interface javato.activetesting.analysis.Analysis) and implement the following methods

```
public interface Analysis {
    public void initialize();
    public void lockBefore(Integer iid, Integer thread, Integer lock);
    public void unlockAfter(Integer iid, Integer thread, Integer lock);
    public void newExprAfter(Integer iid, Integer object,Integer oInv);
    public void methodEnterBefore(Integer iid);
    public void methodExitAfter(Integer iid);
    public void startBefore(Integer iid, Integer parent,Integer child);
    public void waitAfter(Integer iid, Integer thread, Integer lock);
    public void notifyBefore(Integer iid, Integer thread,Integer lock);
    public void notifyAllBefore(Integer iid,Integer thread,Integer l);
    public void joinAfter(Integer iid, Integer parent, Integer child);
    public void readBefore(Integer iid, Integer thread, Long memory);
    public void writeBefore(Integer iid, Integer thread, Long memory);
    public void finish();
}
```

The above methods get called before or after the execution of various instructions of the code under test. For example,

```
public void lockBefore(Integer iid, Integer thread, Integer lock);
```

gets called before thread with unique id "thread" acquires the lock with unique id "lock". Similarly,

```
public void writeBefore(Integer iid, Integer thread, Long memory);
```

gets called before thread with unique id "thread" writes the memory location with address "memory". In each method, the first argument "iid" gives a unique id to each statically inserted method call. The method javato.activetesting.analysis.Observer.getIidToLine(iid) could be used to get a String containing the file name and line number where the method call has been inserted. See javato.activetesting.PrintTraceAnalysis for a simple analysis implementation that simply prints the trace of an execution. Run

```
ant –f run.xml print_trace
```

to test javato.activetesting.PrintTraceAnalysis**. In step 2, you need to implement the class javato.activetesting.HybridAnalysis.** This class should track vector clocks, locks sets, and a database of events and dump the potential races in the file error.log in a suitable format. You need to read error.log in the next step to identify the program locations where you need to pause during active testing. In my implementation of the hybrid race detector, I store a pair of iid to denote a potential race.

In step 3, the instrumented code is run repeatedly. If step 2 reports nRace number of potential race conditions, the instrumented code is run N*nRace times to perform active testing. In our examples, N = 10. The first N runs are run with javato.activetesting.common.Parameters.errorId set to 1, the second N runs are run with javato.activetesting.common.Parameters.errorId set of 2, etc. This number (i.e. javato.activetesting.common.Parameters.errorId) should be used in conjunction with the file "error.log" to get the program locations where you need to pause.

The target "active-loop" in run.xml shows the actions that active testing takes in each N executions of the program under test. The following

```
<antcall target="aactive-loop">
     <param name="javato.activetesting.analysis.class"
     value="javato.activetesting.RaceFuzzerAnalysis"/>
</antcall>
```

shows how to invoke an active tester (RaceFuzzer in this case) by specifying the Java property "javato.activetesting.analysis.class". An analysis class, such as RaceFuzzerAnalysis needs to extend the abstract class javato.activetesting.analysis.CheckerAnalysisImpl, which is almost identical to the class javato.activetesting.analysis.CheckerAnalysisImpl, except that javato.activetesting.analysis.CheckerAnalysisImpl starts a couple of threads that breaks any potential system stalls and livelocks created by the active tester. See section 4 of raced.pdf to learn why such system stalls (i.e. deadlocks) or livelocks could happen. **In step 3, you need to implement the class javato.activetesting.RaceFuzzerAnalysis.** This class needs to use the ActiveChecker class in order to bias the random scheduler.

An active checker is implemented by extending the class javato.activetesting.activechecker.ActiveChecker declared below.

```
public class ActiveChecker {
    final protected void block(int milliSeconds) { … }
    final protected void unblock(int milliSeconds) { … }
    final public static void blockIfRequired() { … }
    public void check(Collection<ActiveChecker> checkers) { block(0);}
```

```
    final public void check() { … }
}
```

An instance of a subclass of ActiveChecker (e.g. you should implement RaceChecker) is equivalent to a transition in Algorithm 1 in activetool.pdf. The check() method defined by this class should be used by RaceFuzzerAnalysis to check if a real race has been created and also to decide whether to pause the current thread or un-pause a paused thread. This method in turn calls the method check(Collection<ActiveChecker> checkers). A subclass of ActiveChecker (such as RaceChecker) should override the check(Collection<ActiveChecker> checkers) method. The argument "Collection<ActiveChecker> checkers" is a list of active checkers (i.e. threads or transitions) that has been paused. An implementation of the method check(Collection<ActiveChecker> checkers) could either block (i.e. pause) the current thread or activate (i.e. un-pause) some other thread that is already paused. In order to pause the current thread, the method block(int milliSeconds) should be called. If "milliseconds" is 0, then the current thread is paused indefinitely; otherwise, the thread is paused for "milliseconds". In order to activate some other instance of ActiveChecker inside the check(Collection<ActiveChecker> checkers) method, unblock(int milliSeconds) should be called on the instance. The target thread is activated after "milliseconds".

Note that the global lock ActiveChecker.lock is used to protect all code that you write. This is done to prevent any data race in your analysis code. Therefore, a call to ActiveChecker.block(0) by a thread while the thread is holding the lock ActiveChecker.lock could cause a system stall. Therefore, a call to ActiveChecker.block(0) does not pause the thread immediately, but records the fact that the thread needs to be paused as soon as it releases the lock ActiveChecker.lock. Therefore, we call the method ActiveChecker.blockIfRequired() after the lock ActiveChecker.lock has been released. If the current thread has been booked to pause then ActiveChecker.blockIfRequired() actually blocks the current thread. See DeadlockFuzzerAnalysis and the comments in RaceFuzzerAnalysis to see how ActiveChecker.blockIfRequired() should be used.

Due Date

March 2, 2009.

How to Submit?

You should email me a link to the entire archive before 11 am on March 3, 2009 along with instructions to run it. Ideally, I will run ant –f run.xml hw2 and see if your implementation reports exactly the same set of races reported by my implementation of RaceFuzzer. Please come to my office on March 4, 2009 afternoon to give me a demo.