**Implementing Lockset based Data Race Detection Technique (Due: May 31, 11:59PM)**

**Overview**

In this assignment, you are asked to implement a Lockset dynamic data race detector for concurrent Java programs, similar to *Eraser*. The lockset algorithm checks that there exists at least one lock that consistently synchronizes accesses to a shared memory address, and detects a data race bug when there is no such lock for a shared memory address. To implement a dynamic analysis technique for Java conveniently, you will use the *CalFuzzer* framework that provides infrastructure for monitoring and analyzing runtime traces of Java programs. You need to demonstrate that your own implementation can detect data races from actual program executions, and analyze in which concurrent execution scenario the data races occur.

**Lockset algorithm**

The lockset algorithm (i.e., Eraser algorithm) is presented in slides 21—28 of the lecture note “Data Race Detection Technique”. Briefly speaking, the algorithm works as follows:

* For each lock/unlock operation, the algorithm should track the set of locks that each thread holds.
* For each memory address, the algorithm maintains the state (i.e., Virgin, Exclusive, Shared, or Shared-Modified), and the set of candidate locks.
* For each memory access operation, the algorithm makes proper transitions in the state of the memory address, updates the candidate lockset, and reports a data race detection if necessary.

**CalFuzzer**

Download the CalFuzzer framework from the course webpage <http://swtv.kaist.ac.kr/courses/cs492b-spring-16/>. This source code contains both tool and target program. Note that this is the updated version of the original CalFuzzer 2.0 distribution, and you must use this version in the assignment.

CalFuzzer is working with Java 7 or higher versions. You need to install Apache Ant (<http://ant.apache.org/>) to build and execute CalFuzzer. There are two Ant scripts build.xml and run.xml under calfuzzer. build.xml is the main script and contains the script for compile and build. The other file run.xml contains the scripts to execute dynamic analyzers (described later). After downloading CalFuzzer 2.1, you can build the framework by the following instruction:

|  |
| --- |
| $ **tar –xvf calfuzzer2.1.tar**$ **cd calfuzzer**calfuzzer$ **ant**…BUILD SUCCESSFULTotal time: 5 seconds |

**LocksetAnalysis**

You have to implement a dynamic analyzer for the Lockset algorithm upon the CalFuzzer framework. The distribution contains the skeleton code for your implementation:

calfuzzer/src/javato/activetesting/analysis/LocksetAnalysis.java.

You should implement the missing parts in the skeleton code (commented as “Implement here”).

Here’s the requirement of your lockset technique:

* For each execution, your technique must print out all alarms of data race detections. Each alarm should contains:
	+ Stack trace of a code location where the data race was detected (i.e., the candidate lockset becomes empty), and
	+ Code location of the last access (read or write) to the memory location before the data race detection.
* Your technique must consider recursive locking because Java programs frequently use it.
* Your technique must not report any data race detection for a volatile memory address
* You may not consider readers-writer lock in this assignment.
* You may not consider the synchronizations other than lock and unlock (e.g., wait, noitify, thread join).

Once you complete LocksetAnalysis.java, you can build your technique by the following command:

|  |
| --- |
| calfuzzer/src/javato/activetesting $ vim LocksetAnalysis.javacalfuzzer/src/javato/activetesting $ cd ../../../calfuzzer $ **ant clean**…BUILD SUCESSFULTotal time: 0 secondscalfuzzer $ **ant**…BUILD SUCCESSFULTotal time: 5 seconds |

The Ant file run.xml contains the scripts to run dynamic analyzers. The script named as lockset-analysis is to instrument a target program, and then execute the instrumented target program with your dynamic analyzer LocksetAnalysis. The script named test\_arraylist1\_lockset executes lockset-analysis for the target program test case ArrayListTest1.java (explained later). You can generate a target test case execution with LocksetAnalysis by the following instructions:

|  |
| --- |
| calfuzzer $ **ant**…BUILD SUCCESSFULTotal time: 5 secondscalfuzzer $ **ant –f run.xml test\_arraylist1\_lockset**… |

**Target programs**

The target program for which you are asked to detect data race is *ArrayList*. The source code is located at src/benchmarks/instrumented/java15/ArrayList.java, and the source code of its super classes is located at the same directory (i.e., src/benchmarks/instrumented/java15).

ArrayList is a library program. A target program using an ArrayList object can call multiple methods of the object concurrently when the object is instantiated with synchronization wrapper Collections. synchronizedList (e.g., List al1 = Collections. synchronizedList (new ArrayList()). Once an ArrayList object is instantiated with synchronization wrapper, users expect that multiple threads can safely execute its methods concurrently.

The test case for ArrayList is src/benchmarks/testcases/ArrayListTest1.java. You must use this program to generate actual executions for data race detection. To detect possible data races, the test case ArrayListTest1.java configures a testing scenario such that there are one ArrayList object with the synchronization wrapper, and 10 concurrent threads each of which invokes one method call.

**What to submit**

You have to submit the followings:

* You have to submit your implementation of LocksetAnalysis.java. Your technique should satisfy all functional requirements described earlier.
* You have to detect data races in ArrayList with the given test case ArrayListTest1.java.

Your report should include the following items.

* The design and implementation of your technique (i.e., LocksetAnalysis.java)
* The bug detection result using your technique with the test case ArrayListTest1.java. Each bug detection result should show (i) the stack trace at the data race detection, and (ii) the code location that accesses the memory address right before the data race detection.

Note that, in your homework grade, the evaluation on your description of the tool design and the result analysis will take a large portion.

**Submission**

Submit your report and result (implementation) via TA’s e-mail by the due date. Please submit the hardcopy of your report to the homework box located at 2nd floor, CS building (E3-1). TA will acknowledge your submission in 12 hours.