Data Race detection in Multi-threaded Java programs

Group No: ***3***
Team Member 1: ***Avanish Kumar Singh (MT14004)***
Contribution of Team Member 1: ***50%*** *approx.*
Team Member 2: ***Ankit Verma (MT14036)***
Contribution of Team Member 2: ***50%*** *approx.*

Data Race

A data race occurs when: two or more threads in a single process access the same memory location concurrently, and at least one of the accesses is for writing, and the threads are not using any exclusive locks to control their accesses to that memory.

Example of Data Race

- The following class (`TargetClass`) has two static integer variables: *`sharedUnsynchronized`* and *`sharedSynchronized.`*
- It also has four static functions:
  - `incrementUnsynchronized()`: It increments the *`sharedUnsynchronized`* variable 100 times. Therefore, the final value should be 100 if the initial value is 0. However, this method is unsynchronized.
  - `decrementUnsynchronized()`: It decrements the *`sharedUnsynchronized`* variable 100 times. Therefore, the final value should be -100 if the initial value is 0. However, this method is unsynchronized.
  - `incrementSynchronized()`: It increments the *`sharedSynchronized`* variable 100 times. Therefore, the final value should be 100 if the initial value is 0. This method is synchronized.
  - `decrementSynchronized()`: It decrements the *`sharedSynchronized`* variable 100 times. Therefore, the final value should be -100 if the initial value is 0. This method is synchronized.

```
public class TargetClass {

	public static int sharedUnsynchronized;
	public static int sharedSynchronized;

	public static void incrementUnsynchronized() {
		for(int i = 0; i < 100; i++) {
			sharedUnsynchronized++;
		}
	}

	public static void decrementUnsynchronized() {
		for(int i = 0; i < 100; i++) {
			sharedUnsynchronized--;
		}
	}

	public static synchronized void incrementSynchronized() {
		for(int i = 0; i < 100; i++) {
			sharedSynchronized++;
```

```
		}
	}

	public static synchronized void decrementSynchronized() {
		for(int i = 0; i < 100; i++) {
			sharedSynchronized--;
		}
	}

}
```

- The following class (`SimpleThreads`) has two fields: *`thread`* and `threadName`.
- It implements Runnable, therefore it can be used to create multiple threads.
- If the current thread is `thread1` then it will call `TargetClass.`*`incrementSynchronized`*`()` and `TargetClass.`*`incrementUnsynchronized`*`()` methods.
- If the current thread is `thread2` then it will call `TargetClass.de`*`crementSynchronized`*`()` and `TargetClass.de`*`crementUnsynchronized`*`()` methods.

```
public class SimpleThreads implements Runnable {

	private Thread thread;
	private String threadName;

	public SimpleThreads(String threadName) {
		this.threadName = threadName;
	}

	public void run() {

		if(threadName.equals("thread1")) {
			TargetClass.incrementSynchronized();
			TargetClass.incrementUnsynchronized();
		}
		else if(threadName.equals("thread2")) {
			TargetClass.decrementSynchronized();
			TargetClass.decrementUnsynchronized();
		}

	}

	public void start() {

		if(thread == null) {
			thread = new Thread(this, threadName);
			thread.start();
		}

	}

}
```

- The following class (`MainClass`) creates two `SimpleThreads` and calls their `start()` method.
- It then prints the values of `TargetClass.`*`sharedUnsynchronized`* and `TargetClass.`*`sharedSynchronized`*.

```
public class MainClass {

	public static void main(String[] args) {

		SimpleThreads thread1 = new SimpleThreads("thread1");
		SimpleThreads thread2 = new SimpleThreads("thread2");
		thread1.start();
		thread2.start();

		try {
			Thread.sleep(1000);
		}
		catch(InterruptedException e) {
			e.printStackTrace();
		}

		System.out.println("sharedUnsynchronized: " +
			TargetClass.sharedUnsynchronized);
		System.out.println("sharedSynchronized: " +
			TargetClass.sharedSynchronized);

	}

}
```

Observations

- Ideally the above code should produce the following output:

```
sharedUnsynchronized: 0
sharedSynchronized: 0
```

- However, following are the outputs that we got after multiple runs:

```
sharedUnsynchronized: -17
sharedSynchronized: 0

sharedUnsynchronized: 0
sharedSynchronized: 0

sharedUnsynchronized: -11
sharedSynchronized: 0
```

- The value of `sharedSynchronized` variable is correct. However, the value of `sharedUnsynchronized` variable is incorrect.
- Moreover, the incorrect value is not consistent. This shows that multiple interleavings of the threads are resulting into multiple values for the `sharedUnsynchronized` variable. This is called data race.
- Data races are usually bad and undesirable. However, its very difficult to prevent or detect data races.

Objective of this project

- The main objective of this project is to detect data races in java multi-threaded programs at runtime using aspects.
- However, the literature-study that we did showed that this problem is ***undecidable***.
- Therefore, our objective is to write an analysis that is *"sound"* and *"as complete as possible"*.

Approach

- We have referred to the following two papers:
  - Eraser: A Dynamic Data Race Detector for Multithreaded Programs by Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro and Thomas Anderson
  - Racer: Aspect-Oriented Race Detection in Java by Eric Bodden and Klaus Havelund
- We have followed the Eraser paper completely and only referred to the Racer paper for understanding the new pointcuts that the authors had introduced in AspectJ specifically for analysing multi-threaded programs.

Brief description of Eraser:

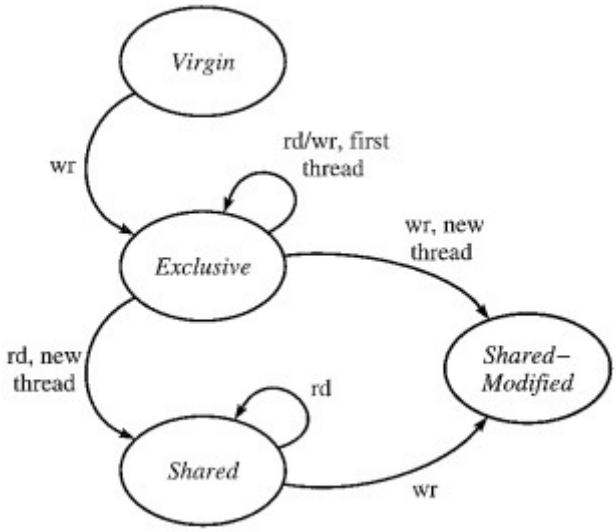
- Eraser employs a "Lock-Set" algorithm.
- Eraser is used to detect data races in C/C++ programs. It is developed specifically to be implemented for analyzing the "C object code".

Brief description of Racer:

- AspectJ did not originally support the analysis of multi-threaded programs.
- The authors of this paper have developed three new pointcuts that can detect multi-threaded paradigms like locking and unlocking.
- As a proof of concept, they implemented the Lock-Set algorithm (used by Eraser) in AspectJ using these new pointcuts.
- They made major modifications in the Lock-Set algorithm. They believed that it is not possible to analyse Java programs by directly implementing the Lock-Set algorithm.
- *However, in this project we have shown that we can implement the Lock-Set algorithm as-it-is without worrying too much about its adaptability, and it works like a charm.*

Eraser's Lock-Set algorithm

Let locksHeldByThread(t) be the set of locks held by thread t.
For each v, initialize locksOnSharedVariable(v) to the set of all locks.
On each access to v by thread t,
set locksOnSharedVariable(v) = locksOnSharedVariable(v) (intersection) locksHeldByThread(t)
if(locksOnSharedVariable(v) == empty), then issue a warning



Eraser keeps the state of all locations in memory. Newly allocated locations begin in the Virgin state. As various threads read and write a location, its state changes according to the transition in the automata above. Race conditions are reported only for locations in the Shared-Modified state.

About the aspects that we built

- We have implemented the lock-set algorithm using aspects.
- The main objective is to detect data races. We are not concerned about how to handle the races.
- *Therefore, this is an "Insertion Security Automata".*
- We allow the code to execute normally. We just print out warnings whenever a potential data race is detected. The warning includes the file-name and the exact line of code where the race is detected.

About the code that we have submitted and the experiments that we performed

We have submitted an Eclipse project with the following packages:

1. **iavanish.analysis:** It contains all the code written for performing the dynamic analysis. It contains two aspects and two helper classes.

2. **iavanish.test1:** It contains the code that we had earlier demonstrated in this project report (refer to the first page). This package is not monitored.

3. **iavanish.test1_checked:** It contains the same code as *iavanish.test1*. This package is monitored. Here's a sample output produced while executing this package (as you can see, all the races that we earlier pointed out have been detected):

```
Race Detected at: iavanish.analysis.RaceDetectedException: TargetClass.java:10
Race Detected at: iavanish.analysis.RaceDetectedException: TargetClass.java:16
Race Detected at: iavanish.analysis.RaceDetectedException: MainClass.java:17
Race Detected at: iavanish.analysis.RaceDetectedException: MainClass.java:18
sharedUnsynchronized: -11
sharedSynchronized: 0
```

4. **iavanish.test1_updated_checked:** We have modified the code of *iavanish.test1* to remove the data races (by following consistent locking). This package is monitored. No warnings are issued when we execute this package.

5. **iavanish.test2:** We have modified the code of *iavanish.test1_updated_checked* in such a way that all accesses to shared variables are performed after acquiring a lock. However, the locking is inconsistent, i.e. it is possible to access a shared variable simultaneously. This package is not monitored.

6. **iavanish.test2_checked:** It contains the same code as *iavanish.test2*. This package is monitored. Here's a sample output produced while executing this package:

```
Race Detected at: iavanish.analysis.RaceDetectedException: TargetClass.java:20
Race Detected at: iavanish.analysis.RaceDetectedException: MainClass.java:19
sharedUnsynchronized: 89
sharedSynchronized: 0
```

7. **iavanish.test3_checked:** The students of the Object Oriented and Programming (OOPD) course, that was offered in the current semester in our college, had written some multi-threaded Java code as part of their assignments, which was available to us (as we are the TAs of this course).
   This package contains the code written by the student who got the highest marks in the respective assignment. We have included this code to show that our aspect not only works on the codes written by us, but also on bigger projects written by others. This package is monitored. Here's a sample output produced while executing this package (it contains 7 data races, we have verified it by reviewing the code manually):

```
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:19
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:20
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:21
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:25
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:26
Race Detected at: iavanish.analysis.RaceDetectedException: ServiceStaff.java:30
Race Detected at: iavanish.analysis.RaceDetectedException: FooLounge.java:40
```

8. **iavanish.test4_checked:** This package contains the code written by the student who got the second highest marks in the respective assignment. This package is monitored. When we analyze this package, it doesn't produce any warning, i.e. no data race is detected.

Conclusions

- The aspects that we developed (by implementing the lock-set) algorithm worked correctly on simple as well as complex code (consisting of many classes and different interleavings).
- The claim made by the authors of the “Racer: Aspect-Oriented Race Detection in Java” paper regarding the im-possibility of directly implementing the Lock-Set algorithm in Java without any modification is not completely true.
- It is possible for our aspect to give false warnings (very rare), however it can never miss a race. Therefore, our analysis is sound.
- Data race is not always unwanted. Some programmers purposely introduce races in their programs for efficiency reasons. Our aspects do not truncate any execution, it just inserts some warnings. Therefore, it does not cause any problem even if the programmer wants the race conditions to exist in his code.
- In the last two experiments, it is proven that even when the code of the student consisted of data races, still she got full marks. That means the TA was not able to catch the data race himself. This proves that detecting data races is very hard.

References

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs by Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro and Thomas Anderson (http://homes.cs.washington.edu/~tom/pubs/eraser.pdf)*
- *Aspect-Oriented Race Detection in Java by Eric Bodden and Klaus Havelund (http://www.havelund.com/Publications/racer-ieee.pdf)*

Code and Artifacts

- We have submitted the complete code and the test cases along with this report.
- The code is also uploaded on GitHub in our public repository at: *https://github.com/iavanish/PAProject*