University of Strathclyde
Computer & Information Sciences

CS313 Assignment
Concurrent Bug Patterns

Filip Lejhanec

Alex Louden

Andrew McGran

Kuba Rudzki

Contents

Various bug patterns in concurrent systems 3
The sleep() bug pattern 3
Non-atomic Operations Assumed to Be Atomic 3
Wrong Lock or No Lock 4
Losing a Notify 5
Prototype Description 5
Chosen metric 5
Implementation 6
Evaluation Process 6
Mutation testing 6
Mutation bugs 6
Remove a signal method call 6
Remove a lock-unlock pair 7
Remove Thread Method Call 7
Shrink critical region 8
Split Critical Region 8
Expand Critical Region 8
Modify method *Timeout* 9
Results 9
Concurrency Coverage 9
Results 10
Conclusion 11
References 12

Various bug patterns in concurrent systems

The sleep() bug pattern

The sleep() bug pattern is a very common one throughout concurrent system development. The pattern is essentially when a developer assumes that the child thread should execute faster than any of its parents' threads. Therefore, because of this assumption the developer adds an appropriate sleep() method to the parent thread in order to try and ensure that the program executes in the desired order i.e. child first, then the parent thread. However the correct way for a developer to ensure the correct order of execution is to use a join() method instead, in order to explicitly wait for the child thread to execute before proceeding to the parent thread(s). This ensures the correct execution due to join() moving onto another thread when another one has finished execution, whereas the sleep() method only invokes a time period to wait before proceeding with the next execution. [1]

Following is a code excerpt from our BankSystem that contains this bug pattern:

```
depositThread.start();
try {
       Thread.sleep(100);
} catch (InterruptedException e) {
       System.out.println("***Senario4: Ohh sleep was disturbed?");
}
System.out.println("***Scenario4: The threads should have been added by now with account
balance of £" + a.balance);
```

A solution would be to replace sleep(100) with join() in order to avoid the sleep failing to wait the allocated time and manifesting the bug. In the banking system project the bug can be observed when the sleep method is used after the withdraw() method and then on the deposit() method. However in this instance join() should have been used in order to ensure the methods are executed in the correct order.

Non-atomic Operations Assumed to Be Atomic

This bug pattern presents itself when a programmer believes that an operation they've used is completed in one step by the resultant bytecode of their program. In reality, the operation is broken down into many steps, which in a concurrent environment, could be interrupted by a context switch. This assumption usually arises when a Java operation only takes up a single line. This visual representation looks like a single step, leading the programmer to falsely believe that it doesn't need any concurrency protection. [1]

```
accountNumber--;
```

The above piece of Java code only takes up one line. The actual sequence of events here can be broken up into the following pseudocode;

- Get accountNumber value from the heap and keep local area copy.
- Decrement it by one.
- Write the new value to the heap.

What seemed like one operation is in fact three steps. The JVM could interrupt a running thread at any point in this sequence, allowing another thread to potentially carry out operations on the same variable.

This could result in two threads both copying the current value to their local copies, modifying the value independently, and then writing their differing values to the heap.

This one misconception can lead to some strange behavior. In initial testing, it could be that this badly-timed context switch never occurs, so the bug never presents itself until the system is in use. The bug will eventually show up and cause issues, and it will be very hard to track down and fix.

Wrong Lock or No Lock

The wrong lock or no lock concurrent system bug is caused by the programmer who either forgets to place a lock in the method of a concurrent code segment or thinks it's unnecessary to place one. Locking wrong part of the code will cause that part of the code segment to be synchronized which actually was not required to be initially, this then leaves some part of the program unsynchronized and exposed to concurrent bugs and race condition occurring.[1] Locking unnecessary bits of code slows their execution by a fraction which in more complicated systems that have lots of wrong locks implemented will have a significant impact on the systems performance. For instance take the banking system created in semester one, there is a deposit method which two or more threads/users can call on the same account.

```
protected void deposit(double amount){
       this.balance += amount;
}
```

While the deposit method is not synchronized and two threads call the deposit method a race condition can occur, the special requirements for it to occur are when the two deposit calls are executed by the JVM at the exact same time. This means that if two threads wanted to increase the account balance which is initially 0 by the amount of 50, instead of finding the balance to be 100 after the two method calls are complete it turns out to be 50. This is how the race condition reveals itself by essentially 'ignoring' one of the method calls since they were executed at the same time. On the other hand wrong lock can mean locking the lock of the bank class instead of a currently accessed account's lock.

```
getBank().getLock().lock()
```

Instead of:

```
lock.lock()
```

The whole system would become very slow as every read/write of the account data like 'withdraw' or 'deposit' would have to wait for the lock of the bank even though two separate accounts would be getting accessed at the same time. This type of wrong lock penalizes every user of the banking system since only one account and only one operation can be performed at the same time. This defeats the purpose of the concurrent systems as all the changes and account accesses have to be performed one by one.

One solution to the problem could be closely analyzing all code fragments that mutate any shared resources and appropriately place a lock around them to ensure all threads use them in the correct manner.

One other possible solution to this concurrent bug is to make sure the locking is done on the right lock of an object making sure you are not locking yourself out by synchronizing the wrong part of the system.

Losing a Notify

A "lost" notify is a bug that is caused by a programmer falsely assuming that a corresponding *wait()* call in thread ***a*** will be executed before a *notify()* call in thread ***b***. Even if the execution in thread ***a*** should happen much sooner than the execution in thread ***b*** it is never guaranteed and the JVM might choose any order of execution. A notify is then fired before thread ***a*** starts to wait and might cause thread ***a*** to not be awakened. [1]

Thread **a**

```
if (amount < balance) {
        wait ();
}
withdraw(balance);
```

Thread **b**

```
wait(10, TimeUnit.SECONDS);
deposit(balance);
notifyAll();
```

Even though in this example the Thread **b** should have taken much longer to get to the notify() statement, it might actually happen earlier that the wait() in Thread **a** and the Thread **b** would get frozen until another interrupt would happen. In this case it would simply wait until another thread would go and deposit, but in other systems the notify might only be one of a kind and the Thread **a** would be permanently frozen.

One of the solutions to this problem is to repeatedly fire a notify() on appropriate actions to ensure that even if the original notify has been missed, there is always another chance for the thread to be woken up. For example if we have a data structure that allows adding and deleting of its elements and it fires a *notifyAll()* on its Condition notEmpty, it can either only fire it when the number of elements changed from 0 to 1 or every time when the number of elements has changed and is not zero. The latter approach would potentially solve the problem. [4]

Another solution is to use a lock that will only become available when thread ***a*** is asleep, and perhaps use a Condition to track whether the thread is asleep or not. Then surround a notify with lock-unlock pair for that lock. This will ensure that the notify will only get called when thread ***a*** is asleep.

Prototype Description

Chosen metric

The concurrency coverage metric that we chose for the project was the define-use (Def-Use) criterion. The interleaving space only achieves complete coverage if all the possible define-use (or read and write) pairs are covered within the prototype. To ensure the correct coverage calculation, a def-use pair must consist of a write from one thread in the program and a read from another thread. In addition, there should be no other write method into the same field in between the read and write pair.

Most of the errors in concurrent systems come from different threads accessing variables at the same time and this causes data racing. Therefore, by testing all the thread accesses we can increase the probability of a data racing bug being uncovered. This metric makes sure that the programmer writing tests focuses on covering field accesses from different threads, making the tests uncover more bugs.

Implementation

We have created an automated coverage calculator as a Java Agent application, making use of the Java Instrumentation class. The Java Agent can be attached to any Java application and monitor its behavior. It is run before the monitored application's class files are loaded and so it can modify the bytecode of this application to inject monitoring code and find all the def-use pairs.

All the field accesses in the application are registered and mix-matched to find all the possible pairs. The field accesses are then injected with extra code to report information when they are executed. We make sure to only inject user defined classes and not java libraries.

The Javassist library was used to simplify the process of retrieving the field accesses and modifying the bytecode. It allowed us to get a mode object-oriented representation of the class files being loaded as well as their methods and even fields.

The dependencies, in our case only the Javassist library, are managed by a Gradle build file.

Evaluation Process

Mutation testing

We are using the BankSystem from semester 1 to introduce the bugs into and will use its existing tests (that had 100% lock-pair coverage) to see how they perform. We will introduce 8 bugs into our system and check if the tests uncover them. There are 86 tests total, all passing at the initial stage of the project:

All 86 tests passed - 1m 30s 111ms

Mutation bugs

Remove Finally Around Unlock

Inside Account in the withdraw method we removed the finally clause around the unlock call.

Original
```
} finally {
        lock.unlock();
}
```
Mutant
```
}//finally                                                                    {
        lock.unlock();
//}
```

All 86 tests passed - 1m 30s 111ms

All 86 tests passed - 1m 30s 127ms

All 86 tests passed - 1m 30s 125ms

All tests run successfully.

Remove a signal method call

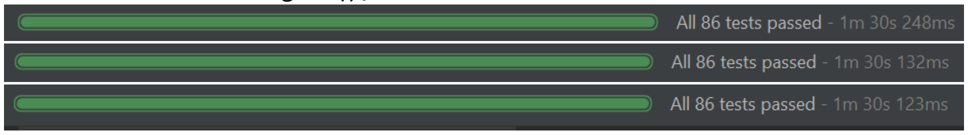
Inside Account in the deposit method we removed the signal call that the balance has been increased.

Original

```
balanceIncreased.signal();
```

Mutant

```
//balanceIncreased.signal();
```



```
All 86 tests passed - 1m 30s 248ms
All 86 tests passed - 1m 30s 132ms
All 86 tests passed - 1m 30s 123ms
```

All tests run successfully.

Remove a lock-unlock pair

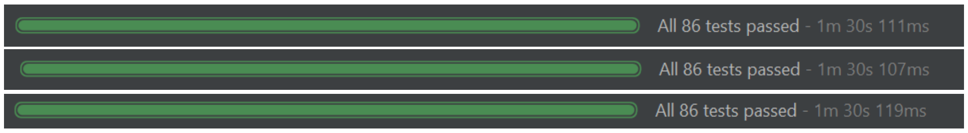
Inside Bank in the addCustomer method we removed the lock and both of its consecutive unlocks.

Original

```
lock.lock();
if (customerDB.containsKey(customer.getID())) {
       lock.unlock();
       System.out.println("This customer already exists!");
       return false;
} else {
       customerDB.put(customer.getID(), customer);
       lock.unlock();
```

Mutant

```
//lock.lock();
if (customerDB.containsKey(customer.getID())) {
       //lock.unlock();
       System.out.println("This customer already exists!");
       return false;
} else {
       customerDB.put(customer.getID(), customer);
       //lock.unlock();
```



```
All 86 tests passed - 1m 30s 111ms
All 86 tests passed - 1m 30s 107ms
All 86 tests passed - 1m 30s 119ms
```

All tests run successfully.

Remove Thread Method Call

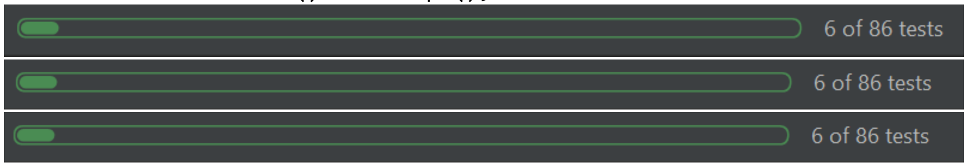
Inside Account in the withdraw method we removed the interrupt call to the current Thread.

Original

```
Thread.currentThread().interrupt();
```

Mutant

```
//Thread.currentThread().interrupt();
```



```
6 of 86 tests
6 of 86 tests
6 of 86 tests
```

All tests run frozen, all counted as failed.

Shrink critical region

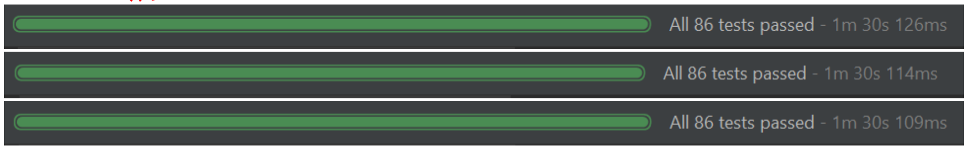
We shifted critical region in Account inside the deposit method, moving lock the call one line ahead.

Original

```
lock.lock();
this.balance += amount;
```

Mutant

```
//lock.lock();
this.balance += amount;
lock.lock();
```


All 86 tests passed - 1m 30s 126ms
All 86 tests passed - 1m 30s 114ms
All 86 tests passed - 1m 30s 109ms

All tests run successfully.

Split Critical Region

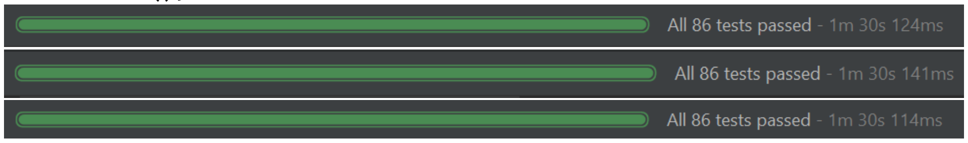
We have split the critical region inside Bank in the openAccount method into two parts.

Original

```
lock.lock();
customer.addAccount(account);
accountDB.put(accountNumber,                                                          account);
lock.unlock();
```

Mutant

```
lock.lock();
customer.addAccount(account);
lock.lock();
lock.unlock();
accountDB.put(accountNumber,                                                          account);
lock.unlock();
```


All 86 tests passed - 1m 30s 124ms
All 86 tests passed - 1m 30s 141ms
All 86 tests passed - 1m 30s 114ms

All tests run successfully.

Expand Critical Region

Inside Bank in the openAccount method call we moved the lock statement 20 lines up.

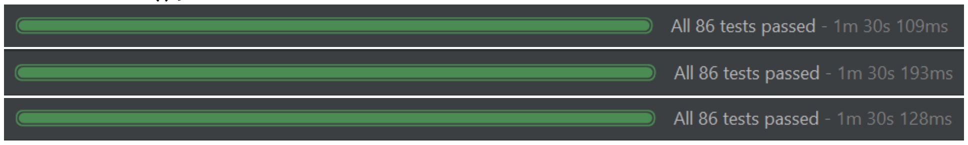
Original

```
customer customer = customerDB.get(customerID);
if (customer == null) return -1; // Customer does not exist
int accountNumber = generateAccountNumber();
Account account;
switch (accountType) {
              ...
}
lock.lock();
```

Mutant

```
lock.lock();
```

```
customer customer = customerDB.get(customerID);
if (customer == null) return -1; // Customer does not exist
int accountNumber = generateAccountNumber();
Account account;
switch (accountType) {
               ...
}
//lock.lock();
```


All 86 tests passed - 1m 30s 109ms
All 86 tests passed - 1m 30s 193ms
All 86 tests passed - 1m 30s 128ms

All tests run successfully.

Modify method *Timeout*

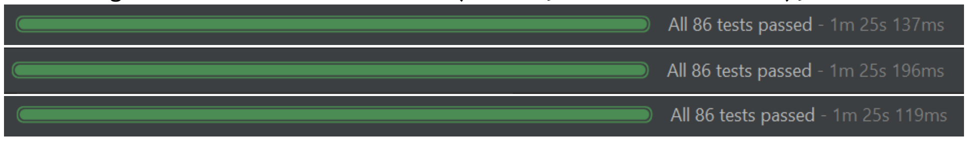
Inside Account in the withdraw method we divided the await time by a factor of 2.

Original

```
isWaiting = balanceIncreased.await(10, TimeUnit.SECONDS);
```

Mutant

```
isWaiting = balanceIncreased.await(10 / 2, TimeUnit.SECONDS);
```


All 86 tests passed - 1m 25s 137ms
All 86 tests passed - 1m 25s 196ms
All 86 tests passed - 1m 25s 119ms

All tests run successfully.

Results

We have run 3 tests for each bug introduced to increase the chance of spotting a bug. The tests have not actually uncovered many bugs, only in one case an error has been spotted. All in all, we have run 24 tests and got an error 3 times. The mutation score is:

(# of bugs revealed) / (total # of bugs) * 100% = mutation score

(1) / (8) * 100% = **12.5%**

We can see that having a 100% coverage does not guarantee that our test is good. As can be seen in our case, we had 100% lock-pair coverage and the only bug that we spotted was an obvious missing notify that froze our whole application.

It is important to write tests that focus on uncovering bugs rather than having 100% coverage, but the coverage can provide a good guideline to the number of tests that we have written.

Concurrency Coverage

We ran the def-use concurrency coverage tests on the BankSystem Scenarios from the semester 1. There are 6 scenarios in total. We will run the tests once at first and then increase the number of runs of all the scenarios to 5, 10 and 20. This will show you the probability increase in having a higher coverage due to the threads interleaving each other and providing more field access pair combinations. There will be 5 executions for each number of runs. The result will be coverage percentage outputted by our monitoring program.

Number of runs = 1

```
1. Covered 12 out of 26
   Total coverage: 46.15%
2. Covered 12 out of 26
   Total coverage: 46.15%
3. Covered 12 out of 26
   Total coverage: 46.15%
4. Covered 12 out of 26
   Total coverage: 46.15%
5. Covered 12 out of 26
   Total coverage: 46.15%
```

Number of runs = 5

```
1. Covered 12 out of 26
   Total coverage: 46.15%
2. Covered 13 out of 26
   Total coverage: 50.00%
3. Covered 13 out of 26
   Total coverage: 50.00%
4. Covered 12 out of 26
   Total coverage: 46.15%
5. Covered 14 out of 26
   Total coverage: 53.85%
```

Number of runs = 10

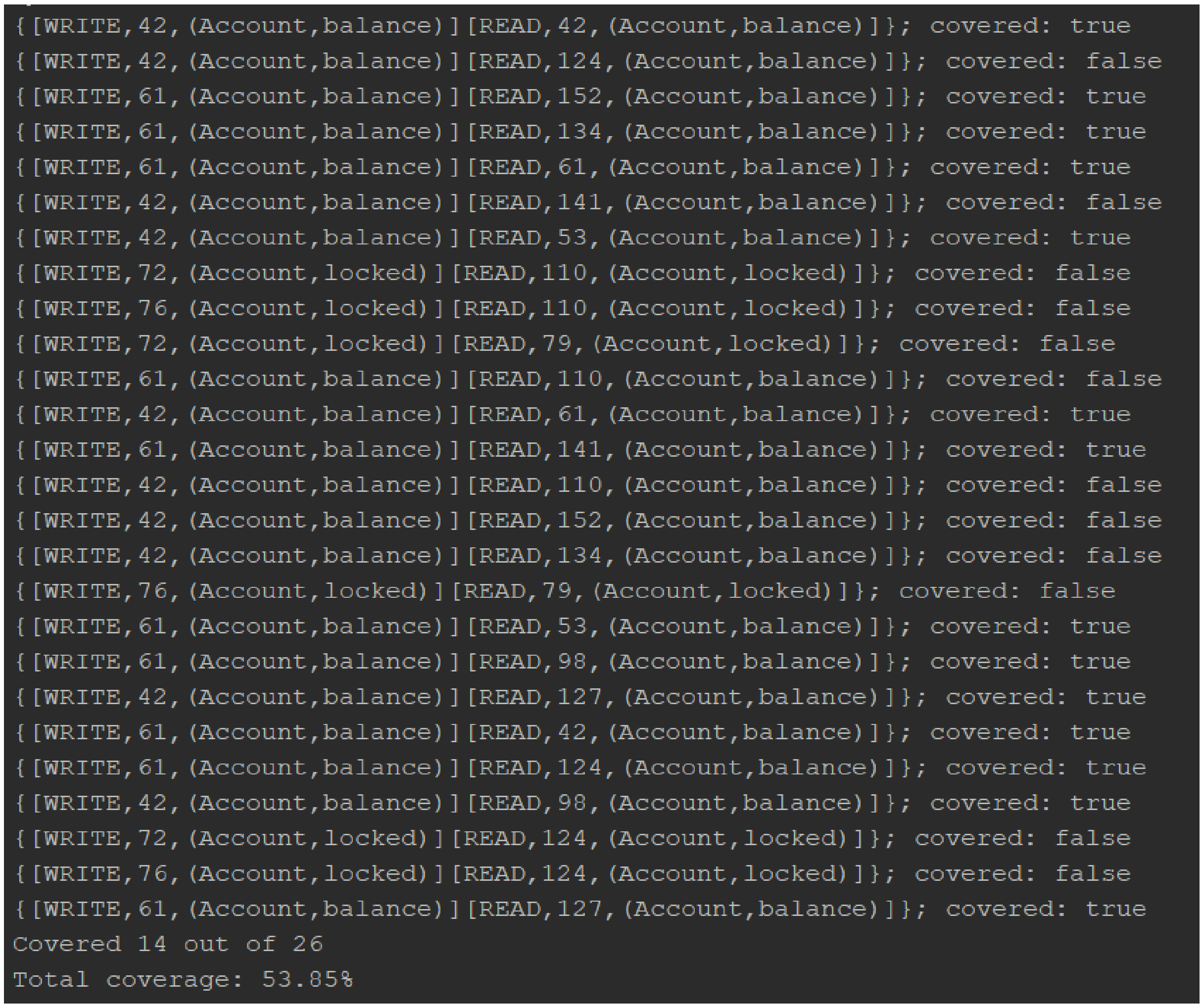
```
1. Covered 13 out of 26
   Total coverage: 50.00%
2. Covered 13 out of 26
   Total coverage: 50.00%
3. Covered 13 out of 26
   Total coverage: 50.00%
4. Covered 13 out of 26
   Total coverage: 50.00%
5. Covered 13 out of 26
   Total coverage: 50.00%
```

Number of runs = 20

```
1. Covered 14 out of 26
   Total coverage: 53.85%
2. Covered 13 out of 26
   Total coverage: 50.00%
3. Covered 13 out of 26
   Total coverage: 50.00%
4. Covered 14 out of 26
   Total coverage: 53.85%
5. Covered 13 out of 26
   Total coverage: 50.00%
```

Results

We have achieved **<u>53.85%</u>** coverage score with our testing tool. The tests are fully automated and can be run on any application. The probability of having a higher coverage increased as the number of runs of all the scenarios increased. The output of the highest coverage result is as follows:



```
{[WRITE,42,(Account,balance)][READ,42,(Account,balance)]}; covered: true
{[WRITE,42,(Account,balance)][READ,124,(Account,balance)]}; covered: false
{[WRITE,61,(Account,balance)][READ,152,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,134,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,61,(Account,balance)]}; covered: true
{[WRITE,42,(Account,balance)][READ,141,(Account,balance)]}; covered: false
{[WRITE,42,(Account,balance)][READ,53,(Account,balance)]}; covered: true
{[WRITE,72,(Account,locked)][READ,110,(Account,locked)]}; covered: false
{[WRITE,76,(Account,locked)][READ,110,(Account,locked)]}; covered: false
{[WRITE,72,(Account,locked)][READ,79,(Account,locked)]}; covered: false
{[WRITE,61,(Account,balance)][READ,110,(Account,balance)]}; covered: false
{[WRITE,42,(Account,balance)][READ,61,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,141,(Account,balance)]}; covered: true
{[WRITE,42,(Account,balance)][READ,110,(Account,balance)]}; covered: false
{[WRITE,42,(Account,balance)][READ,152,(Account,balance)]}; covered: false
{[WRITE,42,(Account,balance)][READ,134,(Account,balance)]}; covered: false
{[WRITE,76,(Account,locked)][READ,79,(Account,locked)]}; covered: false
{[WRITE,61,(Account,balance)][READ,53,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,98,(Account,balance)]}; covered: true
{[WRITE,42,(Account,balance)][READ,127,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,42,(Account,balance)]}; covered: true
{[WRITE,61,(Account,balance)][READ,124,(Account,balance)]}; covered: true
{[WRITE,42,(Account,balance)][READ,98,(Account,balance)]}; covered: true
{[WRITE,72,(Account,locked)][READ,124,(Account,locked)]}; covered: false
{[WRITE,76,(Account,locked)][READ,124,(Account,locked)]}; covered: false
{[WRITE,61,(Account,balance)][READ,127,(Account,balance)]}; covered: true
Covered 14 out of 26
Total coverage: 53.85%
```

Each line contains a write-read pair stating their details (access type, line number, name of the class where the access happened and the name of the method) and whether the pair has been covered.

It is important to note that in some systems (ours for example) it is impossible to achieve 100% def-use coverage since some portions of the code are synchronized. This prevents certain def-use pairs from occurring and will therefore cap the coverage at a fixed maximum value. Despite this fact the def-use coverage seems like a good guideline for testing concurrent systems that helps the programmer to uncover most data-racing bugs.

Conclusion

We have studied several common concurrent bug patterns, the Def-Use concurrency metric, and proposed and implemented a system that calculates coverage under this metric. Using our existing Banking system, we used the prototype to determine the coverage of its existing tests. The prototype system is also flexible, it can be applied to any Java concurrency system to determine its Def-Use coverage.

We've also studied how the Banking system reacts to the bugs we studied being introduced into the system. The challenge of consistently revealing such bugs in a system is clear. Even with mutation testing, the Banking System's tests continued to pass, only failing when one type of bug was introduced.

Under the metric, the system achieves a 53.85% coverage, which seems to be the average figure for Def-Use coverage. About half of the Def-Use pairs were not covered by the tests, so to test more thoroughly, we believe targeting these missing pairs is a good place to start.

References

[1] E. Farchi, Y. Nir and S. Ur, "Concurrent bug patterns and how to test them," *Proceedings International Parallel and Distributed Processing Symposium*, 2003, pp. 7 pp.-.

[2] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. "A study of interleaving coverage criteria." *In The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers (ESEC-FSE companion '07)*. ACM, New York, NY, USA, 533-536.

[3] J. S. Bradbury, J. R. Cordy and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, Raleigh, NC, 2006, pp. 11-11.

[4] Sung-Soo Kim, "Concurrent Bug Patterns," sungsoo.github.io, December 2016, accessed in March 2016 at URL: "http://sungsoo.github.io/2013/12/26/concurrent-bug-patterns.html"