

CONMUTI:
A MUTATION TESTING TOOL FOR CONCURRENT JAVA APPLICATIONS

CALUM MORTIMER

This dissertation was submitted in part fulfilment of requirements for the degree of MSc
Software Development

DEPT. OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF STRATHCLYDE

AUGUST 2020

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes ☒ No ☐

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 7472

I confirm that I wish this to be assessed as a Type 1 2 3 4 5 Dissertation (please circle)

Signature:

A handwritten signature in black ink, appearing to be 'C. M. ...', written over a horizontal line.

Date: 16/08/2020

1. Abstract

The Java programming language provides significant support for concurrent applications. As well as this, there are significant resources and tools available within the Java ecosystem to perform mutation testing. The most notorious of which is the PiTest program [1].

The mutation testing technique evaluates the performance of software test cases by indicating how well test suites fail upon minor alterations to the syntax of the target software.

However, concurrent code presents complex bug patterns. These bug patterns are rarely resolved through the utilisation of conventional mutation testing software and are more difficult to detect than non-concurrent programmer errors.

Bradbury, Cordy and Dingel [2] provided a concise set of mutation operators which could be used to provide comprehensive mutation testing of concurrent Java software.

The subject of this paper, “ConMuti”, is a generic mutation testing tool which was developed to automate the generation and testing of these mutation operators at a source code level for single Java classes.

This paper provides a report on the design of the ConMuti application. The background and justification of the design choices made in the development of the application are discussed, followed by an overview of the detailed design of the application. Finally, an evaluation of the software tool is performed using a selection of concurrent applications.

The results of the evaluation were that the ConMuti application adequately generated a practical set of mutations on target source code. This provided valid and useful data indicating the effectiveness of user test suites on java classes. Furthermore, ConMuti was evaluated positively in its ability to detect potential bugs through improved unit testing.

Further work is required to expand the scope of the automated mutation testing tool to include multiple classes to provide project-level feedback. Further user evaluations (beta tests) are also required to validate all mutation operators and provide further certainty that the application is stable.

Table of Contents

1. Abstract.....	1
2. Introduction	4
3. Background	9
1. Mutation Scoring	9
1. Non-Concurrent Example of Mutation Scoring	9
2. Application to Concurrent Code	11
2. Mutation Operators	12
1. Java Language Syntax.....	12
2. Threads.....	14
3. Explicit Locks	14
4. Semaphores.....	15
5. Latches.....	15
6. Barriers	16
7. Atomic Variables	16
8. Other Concurrency Mechanisms and Mutation Operators.....	16
4. Design	18
1. Source Code Lexical Processing	18
2. Software Design.....	21
1. The Mutation Worker.....	22
2. The MutationCounters and MutationModifiers	23
5. Verification and Validation.....	27
1. Software Engineering Process	27
2. Verification of Project Requirements	28
3. Validation	29
6. Evaluation.....	30
1. Race Condition Source Code	31
2. “account” Project Evaluation	34
3. “almost deadlocked” Project Evaluation	37
4. Evaluation Outcome	38
7. Concluding Remarks	39
8. Scope for Further Work	40
1. Improved Concurrent Object Recognition	40
2. Performance Improvements	40
1. High Latency Activities	40
2. Improved Algorithms	40

3.	Improved Test Selection.....	41
3.	Evaluation.....	41
4.	IDE Integration.....	41
9.	References.....	42
10.	Appendices.....	43
1.	Concurrent Mutation Acronym Guide	43
2.	Concurrent Mutation Acronyms Mapped to Concurrency Bug Patterns.....	44
3.	Requirements List	45
4.	RaceCondition.java Source Code	46
5.	RaceConditionTest.java Source Code	47
6.	AccountProjectTest Account.java Source Code.....	48
7.	AccountProjectTest Main.java Source Code	49
8.	AccountProjectTest ManageAccount.java Source Code	51
9.	AccountProjectTest AccountTest.java Source Code	52
10.	DeadlockProjectTest Source Code	55
11.	DeadlockProjectTest DeadlockTest.java Source Code	56
12.	Evaluation Checklist	57
13.	RaceConditionProjectTest - Evaluation Checklist	60
14.	AccountProjectTest - Evaluation Checklist	63
15.	DeadlockProjectTest - Evaluation Checklist.....	66

2. Introduction

Concurrent software poses significant challenges when it comes to testing and evaluation. The execution order of concurrent threads is inherently unpredictable. This can result in unexpected consequences when programs are deployed. And, exacerbating the severity of the problem, these consequences can be difficult to replicate in a test environment. Rectifying as many oversights in concurrent code as possible prior to release is imperative, especially within concurrent code which is regarded as “mission critical”.

A detailed overview of the concurrent tools available within Java can be found within the language documentation [3].

Mutation testing is one technique software engineers use to determine the effectiveness of a software test. The premise of mutation testing is that if a piece of software is modified in any small way, some element of the software testing strategy should be able to identify this modification. When it comes to small modifications to software, these mutations should be most readily identifiable by the unit tests, as well-written unit tests can more readily cover the space of all class use-cases than any other type of testing. In Java, the mutation operators can be applied to the source code, or at a byte code level.

An example of a Java mutation and an associated JUnit test which “kills” the mutant is shown below:

Original ‘add’ Method (a+b)

```
4 public class MutationExample {
5     public int add(int a,int b){
6         return a + b;
7     }
8
9     @Test
10    public void add(){
11        int a = 1;
12        int b = 1;
13        assertEquals( expected: 2,this.add(a,b));
14    }
15 }
```

Outcome of Unit Test

```
Process finished with exit code 0
```

Mutated ‘add’ Method (a-b)

```
4 public class MutationExample {
5     public int add(int a,int b){
6         return a - b;
7     }
8
9     @Test
10    public void add(){
11        int a = 1;
12        int b = 1;
13        assertEquals( expected: 2,this.add(a,b));
14    }
15 }
```

Outcome of Unit Test

```
java.lang.AssertionError:
Expected :2
Actual   :0
```

Figure 2.1 – The effect of a well-written unit test on mutated source code

In this example, the mutated method ‘int add’ is changed such that (a-b) is returned by the method instead of (a+b). This discrepancy is identified by the unit test, which asserts the ‘int add’ method,

with inputs 1, should return the integer 2. This unit test fails, and consequently, the mutant is “killed” by the unit test.

Running a test suite over all possible mutations of a Java program is extremely labour intensive, or computationally intensive if an automated system is used. Furthermore, conventional mutation testing does not adequately cover use cases relevant to concurrent code.

A challenge posed by concurrent code is the presence of “Bug Patterns”. These are specific, realistic programmer errors which survive the compilation process and are not readily self-evident. An overview of generic bug patterns and how they can be tested is provided in [4].

One of the most obvious examples of a concurrent code bug pattern is “Code Assumed to Be Protected”. This is where the programmer incorrectly assumes that a code segment is protected against interleaving threads.

An example of how the “Code Assumed to Be Protected” bug pattern might manifest itself in the results of a program output are race conditions over synchronized or non-synchronized method calls. The operation “count++”, or post-increment variable *count*, in Java, is inherently non-atomic and uses two operations within the computer CPU scheduler – assignment, followed by increment.

Therefore, if two threads attempt to increment the same value then there is a risk of thread 2 incrementing the old value of *count*, while the post-incremented value of *count* is invisibly held within thread 1. In this scenario, if we assume the starting condition *count* = 0, the outcome would be *count* = 1, instead of two.

Including the “synchronized” keyword in a method call to *count++* would protect against this outcome. This ensures all queued operations to *count* by thread 1 are completed before thread 2’s execution. An example of some code which is protected against this bug is shown below:

```
3      public class RaceCondition {  
4          private int count;  
5  
6          public RaceCondition(){  
7              count = 0;  
8          }  
9  
10         public int getCount(){  
11             return count;  
12         }  
13  
14         public synchronized void increment(){  
15             count++;  
16         }  
17     }
```

Figure 2.2 – A “count++” operation which is well protected from concurrent operations

To illustrate the inadequacy of conventional mutation testing against concurrent code, this small code segment was mutated using a conventional mutation testing tool called “PiTest” [1]. One of the outputs of the PiTest software was a mutation generation report, shown below:

RaceCondition.java

```
1 package com.example.example;
2
3 public class RaceCondition {
4     private int count;
5
6     public RaceCondition(){
7         count = 0;
8     }
9
10    public int getCount(){
11        return count;
12    }
13
14    public synchronized void increment(){
15        count++;
16    }
17 }
```

Mutations

- 1. Substituted 0 with 1 → NO_COVERAGE
- 2. Removed assignment to member variable count → NO_COVERAGE
- 3. Substituted 0 with 1 → NO_COVERAGE
- 4. Substituted 0 with -1 → NO_COVERAGE
- 5. Substituted 0 with 1 → NO_COVERAGE
- 6. Substituted 0 with -1 → NO_COVERAGE
- 1. replaced int return with 0 for com/example/example/RaceCondition::getCount → NO_COVERAGE
- 2. replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE
- 3. Negated integer field count → NO_COVERAGE
- 4. Incremented (a++) integer field count → NO_COVERAGE
- 5. Decrementd (a--) integer field count → NO_COVERAGE
- 6. Incremented (++a) integer field count → NO_COVERAGE
- 7. Decrementd (--a) integer fieldcount → NO_COVERAGE
- 1. Substituted 1 with 0 → NO_COVERAGE
- 2. Replaced integer addition with subtraction → NO_COVERAGE
- 3. Removed assignment to member variable count → NO_COVERAGE
- 4. Negated integer field count → NO_COVERAGE
- 5. Replaced integer operation with first member → NO_COVERAGE
- 6. Replaced integer operation by second member → NO_COVERAGE
- 7. Replaced integer addition with subtraction → NO_COVERAGE
- 8. Replaced integer addition with multiplication → NO_COVERAGE
- 9. Replaced integer addition with division → NO_COVERAGE
- 10. Replaced integer addition with modulus → NO_COVERAGE
- 11. Substituted 1 with 0 → NO_COVERAGE
- 12. Substituted 1 with -1 → NO_COVERAGE
- 13. Substituted 1 with -1 → NO_COVERAGE
- 14. Substituted 1 with 2 → NO_COVERAGE
- 15. Substituted 1 with 0 → NO_COVERAGE
- 16. Incremented (a++) integer field count → NO_COVERAGE
- 17. Decrementd (a--) integer field count → NO_COVERAGE
- 18. Incremented (++a) integer field count → NO_COVERAGE
- 19. Decrementd (--a) integer fieldcount → NO_COVERAGE

Figure 2.3 – Result of the PiTest mutation run on the code in Figure 2.2

These tests were run with the “ALL” selection of the PiTest mutants, rather than the “DEFAULT” mutant selection, and, as shown in Figure 2.3, a series of 32 mutations were generated. These mutants targeted the mathematical arithmetic surrounding the count operations as well as return values of methods. Notably, no attempt was made to remove the “synchronized” keyword of the increment method, which would have resulted in the breakdown of valid concurrent operations on variable *count*. A substantial set of mutations were generated, none of which tested the validity of concurrent code.

It can be advantageous, therefore, to restrict the mutation operators to a set which target concurrency bug patterns. In this project, the scope of the mutation operators is restricted to concurrency mutation operators only at the source code level.

Bradbury, Cordy and Dingel identified a list of concurrency mutation operators for the Java programming language [2]. These mutation operators target concurrency bug patterns within the Java language.

The purpose of the tool developed during this project – “ConMuti” – is to perform mutation testing using these mutation operators at the source code level of any single Java class.

The automated testing is performed by lexical processing and modification of any .java source file using the JavaParser parsing software. This mutated file is rapidly re-compiled using the built-in JavaCompiler and re-tested using JUnit 4 and the provided unit tests. The program output is a mutation report of the .java source file which identifies which mutants applied to the code were killed, and which mutants were not, in combination with a concurrency mutation score. This report can then be used by a software engineer to identify improvements to the engineer’s testing strategy.

3. Background

1. Mutation Scoring

Typically, a mutation testing program scores software in its ability for the unit tests to reject mutants. This mutation score, in its simplest form, is the percentage of mutations which cause a software program's test cases to fail, out of the total number of valid mutations. A valid mutation must be syntactically correct to be included as part of the formula.

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Valid Mutants}} \times 100$$

Equation 3.1 – Mutation score equation

It is possible that a sub-set of the valid mutations behave in the same way as the non-mutated version of the program in every aspect. These mutants are called *Equivalent Mutants*. In such cases, these mutants are counted against the mutation score and in those cases, achieving a 100% mutation score is not possible. Proving empirically whether a mutant *is* equivalent is known as the Equivalent Mutant Problem. This is a significant drawback of mutation testing.

1. Non-Concurrent Example of Mutation Scoring

Introducing the short accompanying test case below with the code introduced in Figure 2.2 improves the mutation score of the non-concurrent example from 0% to 25% by killing 8 of the possible 32 mutants. Refer to the code snippet in Figure 3.1 and the PiTest output in Figure 3.2 for details.

While this is an improvement to the software in guarding against common syntax errors, it still confers no protection to the concurrent bug pattern previously discussed.

```
8 public class RaceConditionTest {
9     @Test
10    public void raceConditionTest(){
11        RaceCondition rc = new RaceCondition();
12        assertEquals( expected: 0,rc.getCount());
13    }
14 }
```

Figure 3.1 – Non-concurrent test case

RaceCondition.java

```
1 package com.example.example;
2
3 public class RaceCondition {
4     private int count;
5
6     public RaceCondition(){
7         count = 0;
8     }
9
10    public int getCount(){
11        return count;
12    }
13
14    public synchronized void increment(){
15        count++;
16    }
17 }
18
19
```

Mutations

1. Substituted 0 with 1 → KILLED
2. Removed assignment to member variable count → SURVIVED
3. Substituted 0 with 1 → KILLED
4. Substituted 0 with -1 → KILLED
5. Substituted 0 with 1 → KILLED
6. Substituted 0 with -1 → KILLED

1. replaced int return with 0 for com/example/example/RaceCondition::getCount → SURVIVED
2. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
3. Negated integer field count → SURVIVED
4. Incremented (a++) integer field count → SURVIVED
5. Decrementd (a--) integer field count → SURVIVED
6. Incremented (++a) integer field count → KILLED
7. Decrementd (--a) integer fieldcount → KILLED

1. Substituted 1 with 0 → NO_COVERAGE
2. Replaced integer addition with subtraction → NO_COVERAGE
3. Removed assignment to member variable count → NO_COVERAGE
4. Negated integer field count → NO_COVERAGE
5. Replaced integer operation with first member → NO_COVERAGE
6. Replaced integer operation by second member → NO_COVERAGE
7. Replaced integer addition with subtraction → NO_COVERAGE
8. Replaced integer addition with multiplication → NO_COVERAGE
9. Replaced integer addition with division → NO_COVERAGE
10. Replaced integer addition with modulus → NO_COVERAGE
11. Substituted 1 with 0 → NO_COVERAGE
12. Substituted 1 with -1 → NO_COVERAGE
13. Substituted 1 with -1 → NO_COVERAGE
14. Substituted 1 with 2 → NO_COVERAGE
15. Substituted 1 with 0 → NO_COVERAGE
16. Incremented (a++) integer field count → NO_COVERAGE
17. Decrementd (a--) integer field count → NO_COVERAGE
18. Incremented (++a) integer field count → NO_COVERAGE
19. Decrementd (--a) integer fieldcount → NO_COVERAGE

Figure 3.2 – Improved mutation score results using the non-concurrent test case

2. Application to Concurrent Code

For this concurrency targeted mutation software, the mutation score is presented as the equivalent to the ordinary mutation score as it applies to concurrency mutations, namely:

$$\text{ConMuti Score} = \frac{\text{Killed Concurrency Mutants}}{\text{Total Valid Concurrency Mutants}} \times 100$$

Equation 3.3 – ConMuti mutation score equation

The outcome of the program should be a shorter set of mutants targeting the concurrent aspects of the code snippet in Figure 2.2, and offering a mutation score specific to concurrency.

Further inspection of the mutants is required to determine whether improvements to the test cases are required or if a mutation is likely to be an Equivalent Mutant. For the ConMuti software, the further detail presented to the user regarding mutation scoring was as follows:

- Mutant state – killed / not killed
- Mutant type (refer to Appendix 1) for details
- Class file name
- Method name (or “null” if mutant is outside of a method)
- Line reference

2. Mutation Operators

A set of mutation operators was defined in [2] which would be used as the basis for this mutation testing tool. This section briefly discusses all the mutation operators implemented and their effects on the behaviour of Java programs. For details of the concurrency acronym definitions, refer to Appendix 1. The concurrency mutation operators can be mapped to the specific concurrency bug patterns via the table provided in Appendix 2. Refer to [4] for a general description of each concurrency bug pattern.

1. Java Language Syntax

Java method call declarations have two optional parameters – static and synchronized – which affect their behaviour in relation to concurrent programs. Method call expressions in Java are non-synchronized by default, which means that all threads can access, and manipulate, the object data simultaneously. Addition of the synchronized keyword protects the object instance against access from a competing thread while the method call is running. Furthermore, addition of the static keyword to a method call declaration protects the entire class instance against competitor threads. The Add Static Keyword (ASTK), Remove Static Keyword (RSTK) and Remove Synchronized Keyword (RSK) mutation operators act on these keywords to indicate whether a program's method calls are correctly verified as thread unsafe / thread safe.

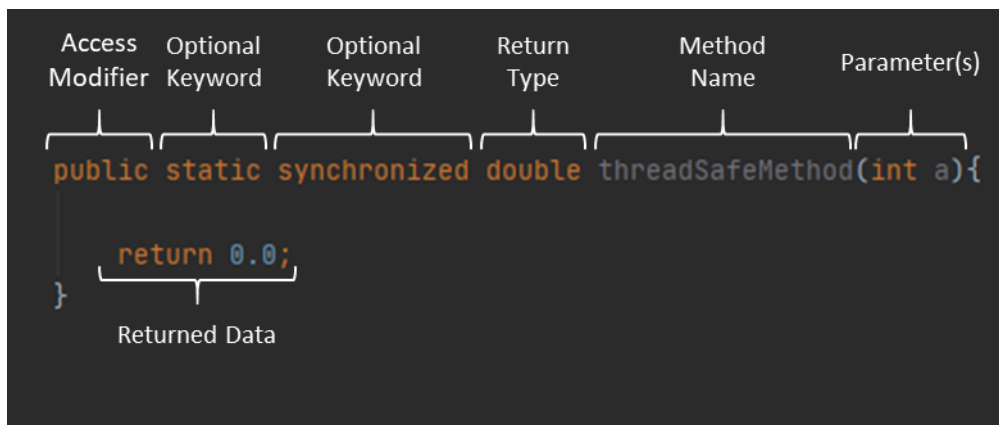


Figure 3.3 – Typical structure of a concurrency method call declaration

Furthermore, the Java language provides the option of synchronizing more specifically particular sets of block statements within methods. These are referred to as synchronized statements / synchronized blocks. The synchronized statement requires a monitor object as a parameter around which to synchronize. Typically, the 'this' keyword is used to synchronize around the current object instance within a class. However, any object may be used.

There are several mutation operators which target the structure of synchronized statements:

The Remove Synchronized Block (RSB) mutation operator removes the synchronicity of a synchronized block altogether.

The Modify Synchronized Parameter (MSP) and Exchange Synchronized Parameters (ESP) mutation operators change the monitor object of synchronized statements to different monitor objects. In the case of MSP, this is changing the monitor to any other valid object. ESP targets the specific case where nested synchronized statements require different monitor objects. These monitor objects are swapped, which may instigate a deadlock bug.

Finally, the Shift Critical Region (SHCR), Shrink Critical Region (SKCR), Expand Critical Region (EXCR) and Split Critical Region (SPCR) mutation operators all attempt to shift the 'critical region' of synchronized statements – i.e. the area of code which is under synchronization. A good testing strategy should confirm that boundary statements – those either side of the synchronized transition – have been correctly attributed to the synchronized or non-synchronized area of code.

```
public double threadUnsafeMethod(int a){  
    //These instructions will execute  
    //without waiting on the object instance  
    synchronized(this){  
        //These instructions will not be  
        //executed until the object instance  
        //is made available  
    }  
    return 0.0;  
}
```

Figure 3.4 – A synchronized block within a non-synchronized method declaration

The volatile keyword is used within variable declarations to guard objects and primitive variables against thread caching. The volatile keyword ensures that data updates to an object are always made in full, in main memory and prohibits the use of caching as an optimisation tool from the Java Virtual Machine (JVM). This means that multiple threads always read the most up-to-date version of an object from memory and objects are not held in an invisible state by another thread. In Java, the 64-bit double and long data types may be altered as 32-bit operations. If a long or double field is declared non-volatile, a 'read' thread may read the first and second half of a variable from different writes.

The Remove Volatile Keyword (RVK) mutation operator removes the volatile keyword from field declarations to expose potential situations where non-atomic operations were assumed to be atomic.

```
private volatile double volatileNumber;
```

Figure 3.5 – A volatile variable

2. Threads

The main class for implementing multithreaded concurrency in Java is the Thread class, which implements the Runnable interface. When a Thread or a Runnable derived class overrides the run() method, this defines the behaviour to be executed when the thread is started. Calling the start() method executes the instruction set held within the run() method declaration.

Threads can be instructed to wait() until notified or until a certain time has elapsed, sleep() for a certain amount of time, join() another thread when it finishes, notify() a waiting thread, notifyAll() waiting threads, or yield() its processor use to the scheduler.

The Modify “X” Time (MXT) mutation operator halves and doubles the time values within the wait, sleep and join method calls, while the Remove Thread “X” Call (RTXC) operator removes any thread method calls.

The Replace Notify All (RNA) operator replaces notifyAll method calls with notify method calls, and the Replace Join with Sleep (RJS) operator replaces join method calls with sleep method calls.

3. Explicit Locks

An explicit lock is any lock which inherits the Lock interface in the Java language. While a monitor object serves as an implicit lock when used as the parameter of a synchronized statement, an explicit lock provides additional behaviour such as “trying” to obtain the lock and performing other behaviour if unavailable.

The Remove Concurrency mechanism “X” Call (RCXC) mutation operator removes lock and unlock method calls to explicit locks, while the Exchange Lock/Permit Acquisition (ELPA) operator changes the acquisition strategy to the lock, i.e. “tryLock” instead of “lock”.

The Replace “X” with Other (RXO) mutation operator changes the lock object a lock or unlock method call is called upon, while the Exchange Explicit Lock Objects (EEO) operator is a variant of the ESP operator for explicit locks – i.e. change the order of acquisition or release of multiple locks.

The Remove Finally around Unlock (RFU) operator removes finally{} blocks from try statements where the finally block surrounds an unlock method called on a lock object. This causes the lock to be retained in some situations where the program exits the current method call.

4. Semaphores

A Semaphore object maintains a set of permits which restrict the number of threads requesting a specific physical or logical resource. The semaphore is initialised with several permits and an optional fairness parameter indicating whether its behaviour is fair or unfair. An unfair semaphore allows barging, where threads can invoke the acquire method to be placed at the top of the queue of waiting threads, whereas a fair semaphore ensures permits are allocated on a First-In-First-Out (FIFO) basis.

This last attribute, the fairness parameter, is altered in the Modify Semaphore Fairness (MSF) mutation operator in order to ensure the fairness of semaphores is tested.

In the Modify “X” Count (MXC) operator, the number of permits available within semaphores is incremented or decremented in order to ensure that tests identify the correct number of permits are being issued by semaphores. The access method calls to semaphores are also incremented or decremented in order to force the semaphore to drain resources or lose performance.

The RCXC operator applied to semaphores removes acquire and release method calls to the semaphore.

Finally, the RXO operator can also be used for semaphores, changing the semaphore object that an acquire or release method is called upon.

5. Latches

The CountdownLatch class objects block one or more threads until one or more other threads have completed their tasks. This is useful if any thread must wait on the completion of multiple threads before continuing. The latch is released after the countdown method has been called a set number of times, or if the optional timeout parameter has been elapsed.

The MXT operator can be applied to latches where the await method specifies a timeout parameter for the latch to count down to zero. In this case, as was the case in the Thread application, the timeout time is changed to double or half of the original set time.

The MXC operator used in semaphores can also be utilised by manipulating the number of threads required to release a latch.

The RCXC operator can be used to remove calls to the countdown method of a latch, and the RXO operator can be used to change latch objects that method calls are attached to.

6. Barriers

The CyclicBarrier class is used when several threads must wait for each other at a common point. Instead of calling countDown, in the case of a latch, each Thread must call the await method on the CyclicBarrier. When the last thread attached to the barrier calls the await method, the barrier is released.

All the mutation operators which can be applied to latches can also be applied to barriers.

One additional mutation operator which can be applied to barriers and not latches is the Modify Barrier Runnable (MBR) operator which acts on the optional Runnable parameter in the CyclicBarrier constructor. This runnable parameter defines a thread which can occur after all threads reach the barrier. If the runnable parameter is removed, the absence of this thread should be detected.

7. Atomic Variables

Classes of variables called atomic variables exist under the Java concurrency package. These ensure that operations on these objects are dealt with atomically while also providing atomic conditional updates using a set of update methods.

One of the methods present within atomic variables is the getAndSet() method which atomically updates an atomic variable while returning its old value. The Switch Atomic with Non-atomic (SAN) mutation operator can be applied to these methods separating them out into separate get() and set() methods, undermining the atomicity of the underlying field.

8. Other Concurrency Mechanisms and Mutation Operators

Other concurrency mechanisms exist in the latest version of the Java language which can be mutated using similar techniques to those defined above. Examples of these include the Phaser,

ExecutorService and Exchanger classes. It is recognized that further mutation operators will be applicable to these classes, but this is deemed to be outside the scope of this project. The design of the ConMuti application encourages expansion to further mutation operators – refer to Section 4.2 for details of the ConMuti software high-level design.

4. Design

1. Source Code Lexical Processing

As aforementioned in the introduction, the purpose of this project was to mutate Java classes at the source code level. For the Java language specifically, this is a design choice. Mutations can readily be applied at the byte code level as an alternative to source code manipulation, making use of Java's interpreted design. PiTest applies mutants at the byte code level. The principal advantage of this strategy is that no Java "compilation" is required – only running the Java Virtual Machine (JVM) against manipulated byte code.

In the context of this application, the set of mutation operators is much smaller as they are focused on concurrency. Furthermore, the target project outcome was to run tests against single classes only. Therefore, it was considered that the enhanced readability of manipulating source code would be of enough benefit to outweigh the performance advantages of byte code modification.

Manipulating source code automatically is a parsing challenge, and there are multiple open-source parsing APIs available. JavaParser [5] was chosen for this project due to its widespread use, strong documentation and thriving support community.

JavaParser creates an Abstract Syntax Tree (AST) of Java source code. The AST is a collection of 'nodes' which are mapped according to their relationships to each other – representing the encapsulation of information within the Java program in a graphical structure.

Changing the source code of a Java program is made possible by converting a source code file into a Compilation Unit (CU) which can then be traversed using Visitor methods and accessing the relationships between AST nodes. For example, in the case of the RFU (Remove Finally around Unlock) concurrency mutator, we first 'visit' a TryStmt which represents a try/catch/finally block. A check is made on the 'finally' block statement to determine if any nodes are unlock() method call expressions. If they are, the 'child' nodes of the finally block are moved to the 'parent' node of the try/catch/finally statement, thereby removing them from finally. Finally, the finally block of the TryStmt is deleted.

A simple Java program and part of its visual representation on the AST is shown overleaf. As shown in the diagram, the AST can become a complex structure even on very small Java applications. However, the visitor methods within the JavaParser library allow structures to be flagged according to their syntactical type. This makes the implementation of some mutation operators very straightforward. In

the example of the RSTK / ASTK mutation operators, it is simply a case of visiting all MethodDeclaration objects within the AST and toggling the isStatic attribute on/off as required.

Upon visiting a syntactical structure such as a MethodDeclaration, the CU is not truncated to the lower nodes below the MethodDeclaration node. Instead, the entire AST is transformed such that all nodes are viewed from the perspective of each MethodDeclaration node. This provides significant flexibility in terms of making modifications to nodes based on their relationships to nodes of a certain type. Casting a CU as another node type can be compared to choosing a different starting point from which to change the AST.

Once the AST has been modified, it can simply be cast to a String and printed to a text file as a new Java program.

```
package com.example.example;

import java.util.concurrent.locks.Lock;

public class SynchronizedCounter {
    Lock l;

    public void ASTExample(){
        try{
            l.lock();
        }
        catch(Exception e){
            System.out.println("lock failure");
        }
        finally{
            l.unlock();
            System.out.println("process finished");
        }
    }
}
```

Figure 4.1 – A typical Java program

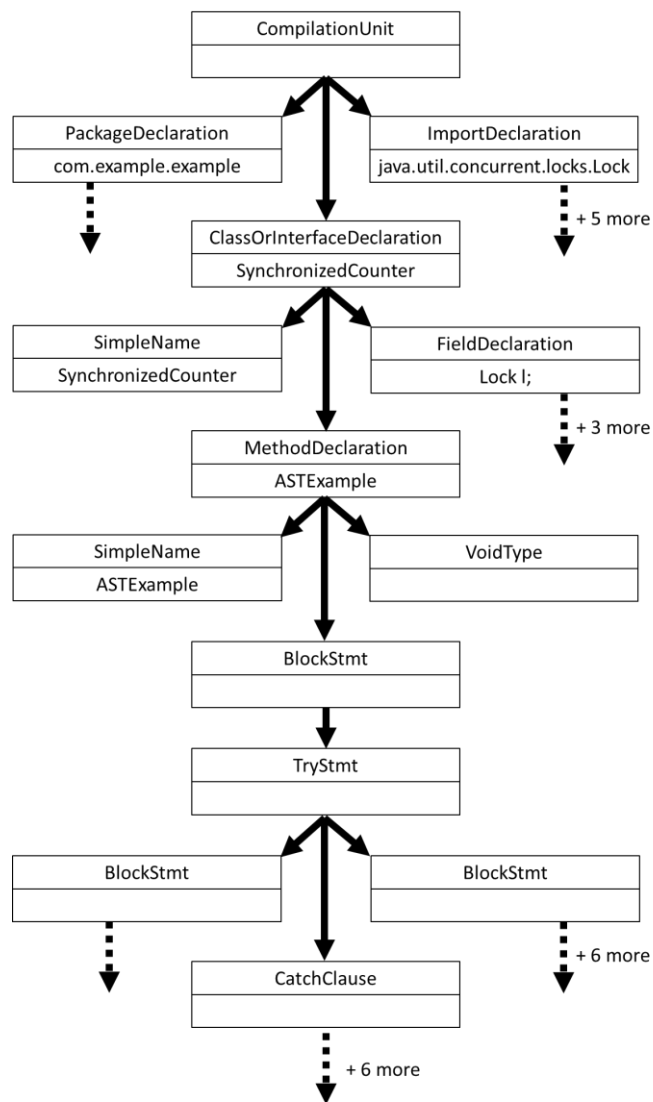


Figure 4.2 – Abstract Syntax Tree (AST) of the program in Figure 4.1

2. Software Design

For full details of the ConMuti system design, refer to the “index.html” file within the attached JavaDoc library and ConMuti source code repository.

The ConMuti software builds upon the JavaParser framework and implements JUnit and the build tools native to the Java language. When the software is run, the core of the program performs a five-step process to generate the mutation scores and the program log output. This process commences within the main method inside the Start class and consists of the following steps:

- Step 1: Delete the log file from the last program run
- Step 2: Obtain the file paths for the following folders:
 - The Java class under test
 - The compiled class file under test
 - The unit test directory
- Step 3: Save a copy of the non-mutated code for back-up
- Step 4: Execute the MutationWorker.run method for each mutation operator to be run against the target source file (refer to Section 4.2.1 – “The Mutation Worker” – for detail)
- Step 5: Replace the source code file with the back-up version

The design of the software prioritises extensibility of mutation operators. All mutation operator classes are inherently polymorphic so that each can be executed by the MutationWorker.run method sequentially without any change to the general behaviour of how mutation operators are executed.

1. The Mutation Worker

As aforementioned, the `MutationWorker` class deals with the execution of a specific type of mutation operator on the target file. The backbone of the class is the `run` method, shown below.

```
public static void run(UserData userData, DirectoryTestRunner directoryTestRunner, MutationCounter counter,
    MutationModifier modifier, String type){
    CompilationUnit cu = getUnmodifiedCU();

    int numberOfMutations = getNoOfMutationOpportunities(cu, counter);

    for (int i=0; i<numberOfMutations; i++){
        cu = getUnmodifiedCU();
        modifier.setMutationsVisited(0);
        modifier.setMutationNumber(i);
        modifier.visit(cu, arg: null);
        saveCU(cu);
        moveMutantToSourceDirectory(userData);
        int compileSuccess = compileMutant(userData);
        if (compileSuccess == 0) {
            moveClassFiles(userData);
            int mutantKilled = directoryTestRunner.runAll();
            DataLog log = new DataLog(mutantKilled, type, userData.getShortJavaFileName(), modifier.getMethodName(),
                modifier.getMutantLineReference());
            logResult(log);
        }
    }
}
```

Figure 4.3 – The `MutationWorker.run` method

The `run` method takes the following parameters as input:

- A `UserData` object which contains the URL locations of all the required files (generated in Step 2 from the core procedure)
- A `DirectoryTestRunner` object which runs JUnit unit tests within a directory and evaluates whether any unit test within that directory has failed
- A `MutationCounter` object. This is the superclass of all mutation counters, each of which count the number of mutation possibilities of that given mutation within a source file.
- A `MutationModifier` object. This is the superclass of all mutation modifiers, each of which modify the Nth mutation within a source file.
- A `String` object defining the acronym for this given mutation type (refer to Appendix X for details)

The `run` method, when called from the `Start` class, executes each extended mutation operator according to the following process:

1. The compilation unit (CU) is generated from the saved target source code file

2. The number of mutation opportunities within the CU is evaluated using the `MutationCounter` object
3. For each mutation opportunity:
 1. The CU is regenerated from the target file
 2. The CU is modified by the `MutationModifier` object
 3. The CU is saved into a “mutant.java” file, which is then moved to replace the target source file in its directory
 4. The mutant is compiled to the target class directory
 5. If the mutant file compiles, i.e. is a “valid” mutation, then a log entry is populated in the “log.txt” file with the mutation details, along with the results of unit tests which are run against the mutation

2. The `MutationCounters` and `MutationModifiers`

The vast majority of the `ConMuti` codebase is populated with classes which inherit either the `MutationCounter` or `MutationModifier` superclasses. These classes behave as follows:

- `MutationCounter`: determines the number of mutations of a given type within a compilation unit
- `MutationModifier`: modifies the Nth mutation of a given type within a compilation unit

Each of these superclasses are further inherited from the `VoidVisitorAdapter` and `ModifierVisitor` classes within the `JavaParser` framework, respectively. An Unified Modelling Language (UML) representation of these relationships is shown in Figure 4.4, below:

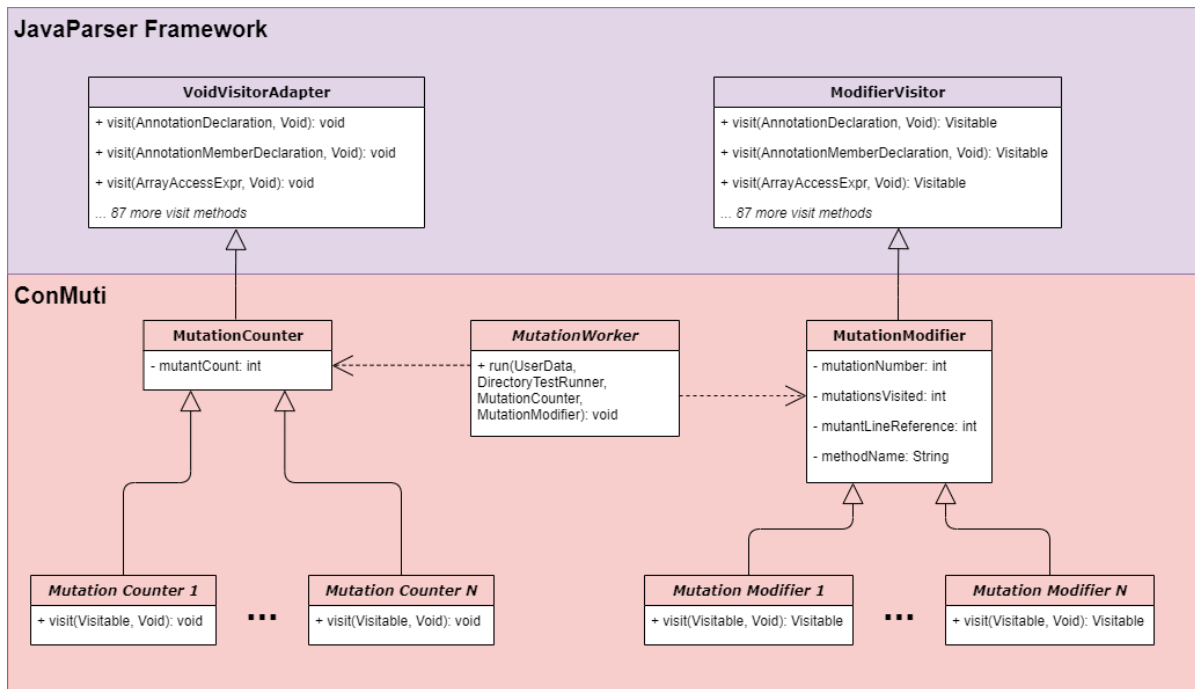


Figure 4.4 – UML representation of the inheritance from the JavaParser framework

The VoidVisitorAdapter and ModifierVisitor classes provide 90 prototype “visit” methods which can be used to visit every type of node within the node passed to the visit method – usually the compilation unit. As aforementioned within section 4.1, all information within a compilation unit can be represented as a node, including the compilation unit itself. The difference between the ModifierVisitor class and the VoidVisitorAdapter class is that the ModifierVisitor may also modify each node within the compilation unit each time it is visited.

When a visit method is called, it is recursively run for each node of the given type it finds within the compilation unit.

Each MutationCounter and MutationModifier object is expected to override one visit method within its grandparent class from the selection of visit methods. For the author of mutation operator classes this provides a high level of flexibility while also enabling maximum code reuse using the JavaParser framework. For example, an author of a mutation operator could choose to cast the compilation unit to a CompilationUnit object within the visit method – this prevents recursion and allows the programmer to manipulate the entire AST. In most cases, ConMuti mutation operator visit methods cast the compilation unit to another type of node and execute a procedure specific to the node type. An example of one MutationModifier visitor within the ConMuti program is shown below:

```

@Override
public MethodDeclaration visit(MethodDeclaration md, Void arg){
    super.visit(md,arg);
    if ((!md.isStatic())&&(md.isSynchronized())){
        if (getMutationsVisited()==getMutationNumber()) {
            md.setStatic(true);
            setMutantLineReference(md.getName().getBegin().get().line);
            setMethodName(md.getDeclarationAsString( includingModifiers: false, includingThrows: false, includingParameterName: false));
        }
        setMutationsVisited(getMutationsVisited()+1);
    }
    return md;
}

```

Figure 4.5 – A MutationModifier visit method for the ASTK mutant

In this example, the visitor method casts the compilation unit to MethodDeclaration types which represent a method declaration within the source code java file. Therefore, the visit method is recursively executed for each method declaration node found within the compilation unit.

This mutation operator – ASTK – adds the static keyword to synchronized methods which are not static.

The *super.visit(md,arg)* method call ensures that all MethodDeclaration nodes which are children of the current MethodDeclaration are also visited.

Next, if the method declaration is identified as being synchronized and non-static, the visitor checks if this is the target method declaration for mutation. If so, the method name and line reference in the MutationModifier superclass are set and the static attribute of the method declaration is set to true.

For each non-static, synchronized method declaration which is visited, the mutationsVisited attribute of the MutationModifier class is incremented.

Finally, whether the method declaration is mutated or not, the MethodDeclaration object is returned by the visitor.

MutationCounters, as opposed to MutationModifiers, simply count all the nodes meeting the set criteria within the compilation unit. In this case no object is returned, but the number of eligible nodes is incremented within the MutationCounter class.

Once all the required MutationCounter and MutationModifier class instances are passed through the MutationWorker run method, the mutation testing of the target java class is completed.

Some mutation operators require more than one `MutationCounter` or `MutationModifier` class, and some mutation operators use other resource classes to maximize code reuse – refer to the ConMuti software documentation for further details.

5. Verification and Validation

1. Software Engineering Process

The development of the ConMuti application was structured around the following process.

A set of 10 requirements was developed (Appendix 3) which would motivate the project outcomes and could be used within the verification process to evaluate the project at all stages of software development.

A high-level design was developed which prioritised the use of existing dependencies (JavaParser, JUnit 4), as well as the extensibility of mutation operators for further development. UML designs were used to manage the dependencies between software classes, an example of which is shown in Figure 4.4.

A detailed design was developed which fulfilled the requirements of mutation operators as defined in [2]. These mutation operators were continuously validated by continuously writing high quality Junit tests for the ConMuti code, as well as being validated against examples of real code.

An evaluation process was conducted (Section 6), which assesses the performance of the ConMuti code against real systems.

2. Verification of Project Requirements

The table below verifies the ConMuti software against the requirements defined in Appendix 3.

Requirement Reference	Description	Demonstration of Compliance
1	The tool will apply concurrency mutators to existing Java source code	Compliant, verified through the evaluation process
2	The tool will run tests/test suites and evaluate the outcomes to infer if a mutant has been killed	Compliant, verified through the evaluation process
3	The tool will be compatible with JUnit 4	Compliant, verified using JUnit 4 unit tests in the evaluation process
4	The tool will be a plug-in for the IntelliJ IDE	Non-Compliant due to project time constraints. However, the modularity of the code enables easy adaption to the IntelliJ framework later. Therefore, compliance with this requirement was de-prioritised during the project process.
5	During mutation testing and evaluation, the user will remain informed of what the application is doing at all stages	Compliant, verified by examining the System.out console output during the evaluation process. Further work could potentially include saving mutants for inspection and review, as per the PiTest use cases.
6	The application should be as fast as reasonably practicable – it shall not be prohibitively time consuming to use	Compliant. Test runs did not take longer than one minute, and the updates provided to the user provide adequate feedback to ensure a crash is not inferred instead of a program delay.
7	The tool will be written in Java	Compliant
8	The tool will perfectly parse concurrent information in Java programs and apply mutation operators – it shall not be confused by extraneous information irrelevant to concurrency	Compliant, verified through unit tests as well as the evaluation process.
9	The tool shall be commented and well documented	Compliant, JavaDoc documents are provided with project software
10	The required input data from the user shall be: Path of source code java file Path of compiled code java file Path to unit test directory	Compliant

Table 5.1 – Analysis of project requirement compliance

3. Validation

The ConMuti software was principally validated using comprehensive JUnit 4 testing of the source code which tested the software's mutation capabilities with different mutation operators and validated that the output of such mutations corresponded to the expected source code.

Further validation was achieved by evaluating the project against the mutation operators in [2]. This tested some of the mutation operators in response to real-life code. A checklist was generated for evaluation purposes, a copy of which is provided in Appendix 12.

6. Evaluation

For the ConMuti tool to be successful in its goal of automated testing of concurrency mutations, it must satisfy the following requirements:

- Satisfactory generation of concurrency mutations as expected by manual inspection
- Evaluation of a significant selection of different concurrency mutations
- Ability to enable the user to reject potential bugs within software
- Killing of mutants only if the test case unit tests are designed to do so

The evaluation of the ConMuti tool consisted of running the tool against three test cases.

For the first test case, as a simple demonstration, the ConMuti tool was run against the race condition example code discussed in the introduction. This would demonstrate both its ability to generate a concurrency mutation and reject the mutation, as well as ignore the 32 potential non-concurrent mutations introduced by the PiTest program.

The second test case was a bugged account manager system selected from a source code repository accessed using an educational licence [6]. This test case was selected randomly from the system in order to demonstrate mutation testing arriving at the successful detection of a program bug.

Finally, ConMuti's ability to reject only those mutants detected by unit tests was demonstrated using a deadlock use case whereby the unit test focused only on rejecting deadlocks. This demonstrated ConMuti's resistance to killing mutants which are not adequately tested by the user's test cases.

Prior to running the projects through ConMuti, a checklist was completed (Appendix 12) which detailed how ConMuti should mutate the selected code. This was achieved by analysing the code in a text editor and manually evaluating the potential mutation opportunities according to [2]. This checklist would also enable analysis of the various parts of the ConMuti software which were tested during the evaluation process.

1. Race Condition Source Code

The evaluation checklist was completed for the RaceCondition class shown in figure 6.1. This checklist is available in Appendix 13. Two mutations were predicted – an invalid ASTK mutation and a valid RSK mutation.

The ConMuti tool was run against the “race condition” example code.

```
3 public class RaceCondition {
4     private int count;
5
6     public RaceCondition(){
7         count = 0;
8     }
9
10    public int getCount(){
11        return count;
12    }
13
14    public synchronized void increment(){
15        count++;
16    }
17 }
```

Figure 6.1 – The RaceCondition example

Without writing any unit test classes to test the functionality of the RaceCondition class, the ConMuti output from the class is shown below in Figure 6.2.

```
1 [0_0],LIVE mutant,TYPE: RSK,FILE: RaceCondition.java,METHOD: void increment(),LINE: 14
2
3
4 Survived mutants found this program run: 1.0
5 Killed mutants found this program run: 0.0
6
7 The total mutation score for the unit tests is 0.0%
8
```

Figure 6.2 – Output of the ConMuti software tool against the RaceCondition example

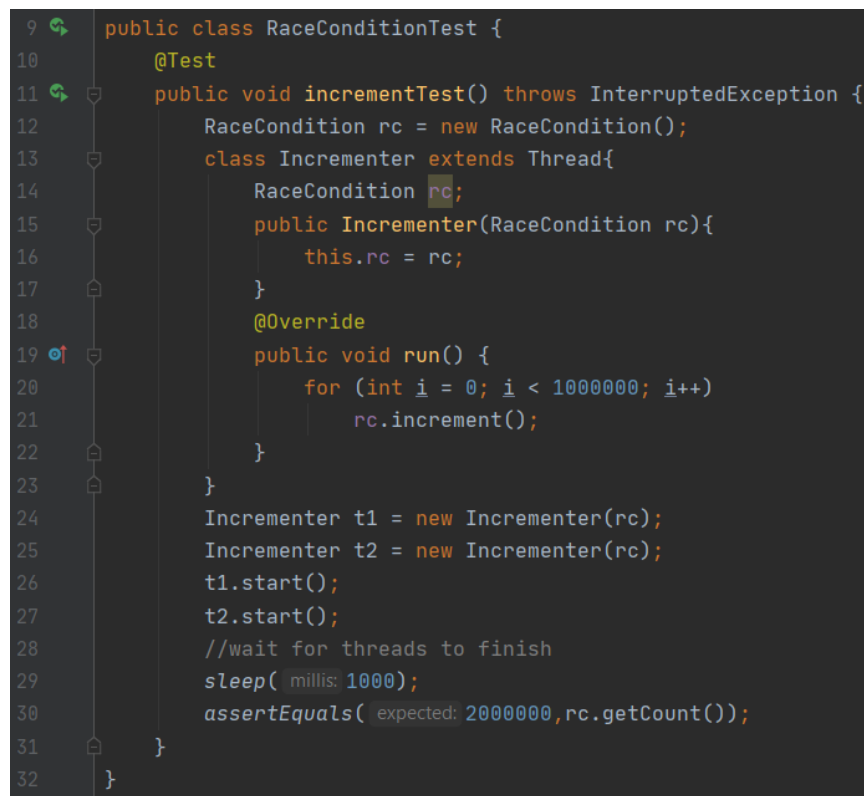
As shown in Figure 6.2, just one possible *concurrency* mutant was applied against the RaceCondition class. This mutant was of type RSK and applied to the “void increment()” method declaration within RaceCondition.java at line 14. In other words, the synchronized keyword was removed, the class successfully compiled, and the unit tests failed to kill the injected mutant. This resulted in a mutation score of 0%.

Debugging the ConMuti tool showed that there was one other candidate mutation – “ASTK” – which would have added a static keyword before the synchronized keyword of the increment method. However, since the field “count” is non-static, this would have failed compilation. Therefore, this candidate mutation was rightfully rejected and did not display in the program output.

These results correspond precisely to those predicted by the evaluation checklist in Appendix 13.

The selection of mutation operators is drastically reduced from the 32 possible mutations imparted by the PiTest mutation testing tool, showing the tool met the requirements of focusing specifically on concurrency mutation operators.

The following test class was added to the test source root of the target project, which would repeatably detect whether the synchronized keyword had been removed from the increment method.



```
9  public class RaceConditionTest {
10     @Test
11     public void incrementTest() throws InterruptedException {
12         RaceCondition rc = new RaceCondition();
13         class Incrementer extends Thread{
14             RaceCondition rc;
15             public Incrementer(RaceCondition rc){
16                 this.rc = rc;
17             }
18             @Override
19             public void run() {
20                 for (int i = 0; i < 1000000; i++)
21                     rc.increment();
22             }
23         }
24         Incrementer t1 = new Incrementer(rc);
25         Incrementer t2 = new Incrementer(rc);
26         t1.start();
27         t2.start();
28         //wait for threads to finish
29         sleep( millis: 1000);
30         assertEquals( expected: 2000000, rc.getCount());
31     }
32 }
```

Figure 6.3 – A class to kill race condition mutants on the increment method

This test procedure ensures that the increment method is treated atomically by threads which attempt to increment the value. Two threads are set to call the increment method one million times. It is almost certain that after two million calls to the increment method, at least one count will have been skipped due to the overlapping execution of the increment method calls. The result of the test with

the *synchronized* keyword present was a pass, while removal of the synchronized keyword results in a test outcome such as the output below in Figure 6.4.

```
java.lang.AssertionError:  
Expected :2000000  
Actual   :1901701  
<Click to see difference>
```

Figure 6.4 – Outcome of a well-designed race condition unit test on a non-synchronized method

Running the ConMuti program against the non-mutated code with the race condition unit test stored in the target unit test directory resulted in the following output:

```
1 [X_X],dead mutant,TYPE: RSK,FILE: RaceCondition.java,METHOD: void increment(),LINE: 14  
2  
3  
4 Survived mutants found this program run: 0.0  
5 Killed mutants found this program run: 1.0  
6  
7 The total mutation score for the unit tests is 100.0%
```

Figure 6.5 – ConMuti output showing defeat of the RSK mutant with improved unit testing

As a result of the improved unit testing, a ConMuti mutation score of 100% is achieved by killing the only mutation applied to the target source code.

2. “account” Project Evaluation

The “account” project from the source code repository [6] was selected as the next project for evaluation against the ConMuti tool. The project consists of three classes which were tested during three runs of the ConMuti program. These Java classes are located within Appendices 6, 7 and 8.

Within the evaluation checklist (Appendix 14), the following mutations were predicted within the ConMuti software output:

Account.java

- Two MSP mutations at line 30 (void transfer()) where a synchronized block locking object is mutated to “this” as well as “System.out”
- An ASTK mutation at line 34 where “static” is added to the print method
- Four RSK mutations on the deposit, withdraw, transfer and print method declarations where the synchronized keyword is removed
- An RSB mutation at line 30 (void transfer()) where a synchronized block is removed

Main.java

- An RTXC mutation at line 36 where a join method call is deleted
- An RJS mutation at line 36 where a join method call is replaced with sleep(10000)

ManageAccount.java

- No mutations

The results of the ConMuti tool output were as follows:

```
1 [0_0],LIVE mutant,TYPE: MSP (sync on 'this'),FILE: Account.java,METHOD: void transfer(Account, double),LINE: 30
2 [0_0],LIVE mutant,TYPE: MSP (sync on System.out),FILE: Account.java,METHOD: void transfer(Account, double),LINE: 30
3 [0_0],LIVE mutant,TYPE: ASTK,FILE: Account.java,METHOD: void print(),LINE: 34
4 [0_0],LIVE mutant,TYPE: RSK,FILE: Account.java,METHOD: void deposite(double),LINE: 13
5 [0_0],LIVE mutant,TYPE: RSK,FILE: Account.java,METHOD: void withdraw(double),LINE: 17
6 [0_0],LIVE mutant,TYPE: RSK,FILE: Account.java,METHOD: void transfer(Account, double),LINE: 21
7 [0_0],LIVE mutant,TYPE: RSK,FILE: Account.java,METHOD: void print(),LINE: 34
8 [0_0],LIVE mutant,TYPE: RSB,FILE: Account.java,METHOD: void transfer(Account, double),LINE: 30
9
10
11 Survived mutants found this program run: 8.0
12 Killed mutants found this program run: 0.0
13
14 The total mutation score for the unit tests is 0.0%
```

Figure 6.6 – ConMuti output from the “Account.java” class within the “account” project

```

1 [0_0],LIVE mutant,TYPE: RJS,FILE: Main.java,METHOD: void main(String[]),LINE: 36
2
3
4 Survived mutants found this program run: 1.0
5 Killed mutants found this program run: 0.0
6
7 The total mutation score for the unit tests is 0.0%
8

```

Figure 6.7 – ConMuti output from the “Main.java” class within the “account” project

```

1
2
3 Survived mutants found this program run: 0.0
4 Killed mutants found this program run: 0.0
5
6 Cannot evaluate the mutation score as no mutations were found!
7

```

Figure 6.8 – ConMuti output from the “AccountManager.java” class within the “account” project

As shown in figures 6.6 through 6.8, all mutations predicted through the evaluation checklist were shown within the ConMuti output results, except for one potential mutation which was the RTXC mutation within the Main.java file. This was not processed in the ConMuti output because the java file was syntactically incorrect: no InterruptedException could be thrown within the try clause since there were no statements (i.e. join or sleep) which could throw an InterruptedException. Therefore, the predicted mutation was not valid.

Some tests were written to address the mutations within the Account.java class – these tests are shown in Appendix 9. One of the tests, “transferRace2”, was designed to confirm that a third bank account would register the correct amount when two other bank accounts were attempting simultaneous transfers from different threads. The original software failed the test – this was due to inadequate protection of the receiving account around the identified synchronized block in Account.java. The following change was proposed:

```

30 //Bugged Code
31 //synchronized (ac) { ac.amount+=mn; }
32 //New Code
33 ac.depsite(mn);

```

Figure 6.9 – Updated Account.java code segment in response to mutation testing evaluation

This updated, non-bugged version of the software used the inherent synchronization of the receiving account's deposit method rather than attempting an explicit lock on the object.

No tests were written to address the RJS mutation operator within the "Main" class, as this sleep bug would have been extremely difficult to instrument with a unit test due to the fast execution of threads in this example.

The results of the ConMuti evaluation tool on "Account.java" with the inclusion of the "AccountTest" are shown below.

```
1 [0_0],LIVE mutant,TYPE: ASTK,FILE: Account.java,METHOD: void print(),LINE: 37
2 [X_X],dead mutant,TYPE: RSK,FILE: Account.java,METHOD: void depsite(double),LINE: 13
3 [X_X],dead mutant,TYPE: RSK,FILE: Account.java,METHOD: void withdraw(double),LINE: 17
4 [X_X],dead mutant,TYPE: RSK,FILE: Account.java,METHOD: void transfer(Account, double),LINE: 21
5 [0_0],LIVE mutant,TYPE: RSK,FILE: Account.java,METHOD: void print(),LINE: 37
6
7
8 Survived mutants found this program run: 2.0
9 Killed mutants found this program run: 3.0
10
11 The total mutation score for the unit tests is 60.0%
12
```

Figure 6.10 – Updated ConMuti output from the "Account.java" class within the account project

As shown in Figure 6.10 – the inclusion of unit tests reduced the number of mutations through the removal of a synchronized block bug. As well as this, a mutation score of 60% was achieved by the new unit tests. The outcome of the mutation testing, and the authoring of unit tests in response to the mutation testing, was a more robust Account class within the "account" project. Furthermore, the ASTK and RSK on void print() mutations could not be eliminated as the "print()" method was empty. Therefore, this mutation score could be considered as 100% when accounting for equivalent mutants. There is a high probability that the account program is now bug free.

3. “almost deadlocked” Project Evaluation

An application was written which was designed to simulate a bug-free project with multiple nested synchronized statements (Appendix 10). The synchronized statements, if reversed by an ESP mutation, would cause a classic deadlock bug within the source code. A unit test was written which would ascertain whether the class under test was deadlocked (Appendix 11). The expectation checklist of the mutants generated by the ConMuti tool are listed in Appendix 15.

Since the unit test is only designed to eliminate bugs associated with deadlocking of synchronized statements, other mutations (e.g. an RSB mutation) should be expected to survive the ConMuti process in this example. Only those which cause explicit deadlocking of the program should be killed – thus demonstrating the effectiveness of ConMuti in killing only those mutants which are rejected by unit tests. The output of the ConMuti application is shown below.

```
1 [0_0],LIVE mutant,TYPE: MXT (double time value),FILE: Process1.java,METHOD: void run(),LINE: 13
2 [0_0],LIVE mutant,TYPE: MXT (half time value),FILE: Process1.java,METHOD: void run(),LINE: 13
3 [0_0],LIVE mutant,TYPE: MSP (sync on 'this'),FILE: Process1.java,METHOD: void run(),LINE: 17
4 [0_0],LIVE mutant,TYPE: MSP (sync on 'this'),FILE: Process1.java,METHOD: void run(),LINE: 11
5 [0_0],LIVE mutant,TYPE: MSP (sync on System.out),FILE: Process1.java,METHOD: void run(),LINE: 17
6 [0_0],LIVE mutant,TYPE: MSP (sync on System.out),FILE: Process1.java,METHOD: void run(),LINE: 11
7 [X_X],dead mutant,TYPE: ESP,FILE: Process1.java,METHOD: void run(),LINE: 17
8 [0_0],LIVE mutant,TYPE: RSB,FILE: Process1.java,METHOD: void run(),LINE: 11
9 [0_0],LIVE mutant,TYPE: RSB,FILE: Process1.java,METHOD: void run(),LINE: 11
10 [0_0],LIVE mutant,TYPE: SPCR,FILE: Process1.java,METHOD: void run(),LINE: 11
11
12
13 Survived mutants found this program run: 9.0
14 Killed mutants found this program run: 1.0
15
16 The total mutation score for the unit tests is 10.0%
17
```

Figure 6.11 – ConMuti output from the “almost deadlocked” DeadlockProjectTest example

Figure 6.11 shows the ConMuti output registering a score of 10% due to the rejection of only the ESP mutation which caused a deadlock bug to flag the unit test. Other mutations could survive. The output of the ConMuti tool corresponds to the results which were expected using the evaluation checklist.

4. Evaluation Outcome

Using the evaluation checklists it was shown that ConMuti successfully generated concurrency mutations in some of the implemented use cases. Out of the 24 implemented mutation operators, examples of mutations 1, 2, 3, 7, 10, 13, 15, 16 and 24 in the evaluation checklists were highlighted by the use cases. While this is only a subset of the implemented operators, a significant diversity of mutation operators was tested. Operators which were not covered in these examples are validated through the project unit tests only.

It was also demonstrated that ConMuti could successfully kill mutants based on user provided unit tests. And, on one occasion, it was shown that ConMuti could direct a user to identify bugs in the test case source code. Thus, the evaluation outcomes were achieved.

7. Concluding Remarks

In this paper it has been shown how ConMuti can be utilized as a tool to produce better concurrent Java software. Through evaluation of three test cases it was shown to be an effective tool which produced results which delivered value to the test case authors.

However, in developing the project some significant flaws within the design were identified. Significant further work on the tool is required prior to its presentation as a fit for purpose application. Some of these improvements are discussed within the next section.

8. Scope for Further Work

1. Improved Concurrent Object Recognition

ConMuti detects concurrent method expressions through their text using the AST. Methods such as `wait()`, `sleep()` and `join()`, are eliminated in the RTXC mutation operator based purely on their name. This was a design decision formed in the early design of ConMuti.

A further check is required to establish if the methods are really called upon Runnable objects, as the method names are not protected from use by other classes unrelated to concurrency. However, as the Thread object and Runnable interfaces themselves are extendable, significantly more AST manipulation would be required to determine which objects are / are not related to concurrency. If this is not addressed, then spurious mutations on non-concurrent methods which share concurrent method names would be inevitable.

2. Performance Improvements

1. High Latency Activities

The program also relies heavily on the computer long-term memory to move the mutated source code files into the target directory for the unit tests to be executed. An alternative approach would be to run unit tests against mutants as they are held in runtime memory as opposed to disk storage.

2. Improved Algorithms

The extension of the MutationModifier and VoidVisitorAdapter classes to implement mutation operators comes with some sub-optimal iteration through program nodes. Firstly, the entire AST is traversed before any mutations are modified purely for the purpose of identifying how many potential mutations and therefore method calls are made to the MutationModifier within the MutationWorker.run method. If a bespoke design was created where an AST could be traversed as it mutates while “counting down” to the end of the file this would be more efficient.

Furthermore, the MutationModifier iterates through all target nodes of the AST, regardless of whether they should be mutated. After a mutation is applied then the MutationModifier traverses the rest of the nodes without applying any changes. This is a waste of execution space and could be improved by only traversing relevant target nodes and breaking the execution when a mutation is applied.

3. Improved Test Selection

The testing logic of the ConMuti software was chosen to be simplistic when weighed up against the time available for development. All tests within a user directory are run when a mutation is generated, even if the tests are completely irrelevant to the mutation.

PiTest optimises JUnit tests in order that only relevant tests are executed against mutated code. This confers a significant advantage. The ConMuti software could be significantly improved if a logical testing strategy is built into the design.

3. Evaluation

As discussed in Section 6, only a subset of mutation operators was evaluated through the use case selections on this occasion. Many more use cases for a diverse set of mutation operators should be identified to increase confidence in the tool.

4. IDE Integration

ConMuti is a modular Java system which is readily portable into an IntelliJ or Eclipse IDE plugin. The inputs to ConMuti are directory locations which could be more easily selected using file choosers rather than text input.

9. References

- [1] Pitest, [Online]. Available: <https://pitest.org/>. [Accessed 16 July 2020].
- [2] J. S. Bradbury, J. R. Cordy and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," *Second Workshop on Mutation Analysis*, pp. 1-10, 7 November 2006.
- [3] Oracle Corporation, "Essential Classes: Concurrency," [Online]. Available: <https://docs.oracle.com/javase/tutorial/essential/concurrency/>. [Accessed 16 July 2020].
- [4] E. Farchi, Y. Nir and S. Ur, "Concurrent Bug Patterns and How to Test Them," *17th International Parallel and Distributed Processing Symposium*, 2003.
- [5] JavaParser, [Online]. Available: <https://javaparser.org/>. [Accessed 16 July 2020].
- [6] National Science Foundation, "Software-artifact Infrastructure Repository," [Online]. Available: <https://sir.csc.ncsu.edu/>. [Accessed 15 August 2020].

10. Appendices

1. Concurrent Mutation Acronym Guide

An acronym list for concurrent mutant taxonomy was proposed in [2]. The following descriptions are slightly modified in order to support the Java language taxonomy used within the JavaParser library.

The acronyms are defined alphabetically in this list rather than attributed to their respective sub-categories as defined in [2].

ASTK – Add static keyword to method declaration
EELO – Exchange explicit lock objects
ELPA – Exchange locking or semaphore permit acquisition strategy rules (e.g. lock() -> tryLock())
ESP – Exchange synchronized statement parameters (swap locking objects between nested blocks)
EXCR – Expand critical region
MBR – Modify barrier runnable parameter
MSF – Modify semaphore fairness parameter
MSP – Modify synchronized statement parameter (locking object)
MXC – Modify concurrency mechanism permit/thread counts
MXT – Modify method call expression time value
RCXC – Remove lock, semaphore, barrier or latch method call expressions
RFU – Remove finally keyword of try/catch statements around unlock method call expressions
RJS – Replace join() method call expression with sleep() method call expression
RNA – Replace notifyAll() method call expression with notify()
RSB – Remove synchronized block (remove the synchronicity of a synchronized statement)
RSK – Remove synchronized keyword from method declaration
RSTK – Remove static keyword from method declaration
RTXC – Remove thread method call expressions
RVK – Remove volatile keyword from declarations
RXO – Replace concurrency method call expression object with a different concurrency object
SAN – Switch atomic call with non-atomic (i.e. AtomicInteger.getAndSet(int) becomes get then set)
SHCR – Shift critical region
SKCR – Shrink critical region
SPCR – Split critical region

2. Concurrent Mutation Acronyms Mapped to Concurrency Bug Patterns

Table copied from Mutation Operators for Concurrent Java [2].

Concurrency Bug Pattern	Mutation Operators
Nonatomic operations assumed to be atomic bug pattern	RVK, SAN
Two-stage access bug pattern	SPCR
Wrong lock or no lock bug pattern	MSP, ESP, EELO, SHCR, SKCR, EXCR, RSB, RSK, ASTK, RSTK, RCXC, RXO
Double-checked locking bug pattern	-
The sleep() bug pattern	MXT, RJS, RTXC
Losing a notify bug pattern	RTXC, RCXC
Notify instead of notify all bug pattern	RNA
Other missing or non-existent signals bug pattern	MXC, MBR, RCXC
A “blocking” critical section bug pattern	RFU, RCXC
The orphaned thread bug pattern	-
The interference bug pattern	MXT, EXCR, EELO, RXO
The deadlock (deadly embrace) bug pattern	MSF, ELPA
Resource exhaustion bug pattern	MXC
Incorrect count initialization bug pattern	MXC

3. Requirements List

Requirement Reference	Description
1	The tool will apply concurrency mutators to existing Java source code
2	The tool will run tests/test suites and evaluate the outcomes to infer if a mutant has been killed
3	The tool will be compatible with JUnit 4
4	The tool will be a plug-in for the IntelliJ IDE
5	During mutation testing and evaluation, the user will remain informed of what the application is doing at all stages
6	The application should be as fast as reasonably practicable – it shall not be prohibitively time consuming to use
7	The tool will be written in Java
8	The tool will perfectly parse concurrent information in Java programs and apply mutation operators – it shall not be confused by extraneous information irrelevant to concurrency
9	The tool shall be commented and well documented
10	The required input data from the user shall be: Path of source code java file Path of compiled code java file Path to unit test directory

4. RaceCondition.java Source Code

```
package com.example.example;

public class RaceCondition {
    private int count;

    public RaceCondition() {
        count = 0;
    }

    public int getCount() {
        return count;
    }

    public synchronized void increment() {
        count++;
    }
}
```


5. RaceConditionTest.java Source Code

```
package com.example.example;

import org.junit.Test;

import static java.lang.Thread.sleep;
import static org.junit.Assert.*;

public class RaceConditionTest {
    @Test
    public void incrementTest() throws InterruptedException {
        RaceCondition rc = new RaceCondition();
        class Incrementer extends Thread{
            RaceCondition rc;
            public Incrementer(RaceCondition rc){
                this.rc = rc;
            }
            @Override
            public void run(){
                for (int i=0;i<1000000;i++)
                    rc.increment();
            }
        }
        Incrementer t1 = new Incrementer(rc);
        Incrementer t2 = new Incrementer(rc);
        t1.start();
        t2.start();
        //wait for threads to finish
        sleep(1000);
        assertEquals(2000000,rc.getCount());
    }
}
```

6. AccountProjectTest Account.java Source Code

```
package com.example.example;//import java.lang.*;

public class Account {
    double amount;
    String name;

    //constructor
    public Account(String nm,double amnt ) {
        amount=amnt;
        name=nm;
    }
    //functions
    synchronized void deposite(double money){
        amount+=money;
    }

    synchronized void withdraw(double money){
        amount-=money;
    }

    synchronized void transfer(Account ac,double mn){
        amount-=mn;
        //System.out.println("ac.amount is $" +ac.amount);
        if (name.equals("D")) {
            System.out.println("unprotected");
            ac.amount+=mn;//no aquire for the other lock!!
            //+= might cause problem --it is not atomic.
        } else {
            //System.out.println("protected");
            //Bugged Code
            //synchronized (ac) { ac.amount+=mn; }
            //New Code
            ac.deposite(mn);
        }
    }

    synchronized void print(){
    }

} //end of class Account
```

7. AccountProjectTest Main.java Source Code

```
package com.example.example;

/**
 * Title:      Software Testing course
 * Description: The goal of the exercise is implementing a program which
demonstrate a parallel bug.
 * In the exercise we have two accounts.The program enable tranfering
money from one account to the other.Although the functions were defended by
locks (synchronize) there exists an interleaving which we'll experience a
bug.
 * Copyright:  Copyright (c) 2003
 * Company:    Haifa U.
 * @author Zoya Shaham and Maya Maimon
 * @version 1.0
 */

public class Main {

    public static void main(String[] args) {

        try{
            ManageAccount.num = 5;
            ManageAccount[] bank=new ManageAccount[ManageAccount.num];
            String[] accountName={new String("A"),new String("B"),new
String("C"),new String("D"),new String("E"),
                                new String("F"),new String("G"),new String("H"),new
String("I"),new String("J"),};
            for (int j=0;j<ManageAccount.num;j++){
                bank[j]=new ManageAccount(accountName[j],100);
                ManageAccount.accounts[j].print();//print it
            }

            //start the threads
            for (int k=0;k<ManageAccount.num;k++){
                bank[k].start();
            }

            // wait until all are finished
            for (int k=0;k<ManageAccount.num;k++){
                bank[k].join();
            }
            ManageAccount.printAllAccounts();

            //updating the output file
            boolean bug = false;
            //flags which will indicate the kind of the bug
            for (int k=0;k<ManageAccount.num;k++){
                //System.out.println("account "+k+" has
$"+com.example.example.ManageAccount.accounts[k].amount);
                if (ManageAccount.accounts[k].amount<300) {
                    bug=true;
                }
                else if (ManageAccount.accounts[k].amount>300) {
                    bug=true;
                }
            }
        }
    }
}
```

```
        if (bug)
            throw new RuntimeException("bug found");

    } catch (InterruptedException e) {
    }

    } //end of function main
} //end of class com.example.example.Main
```

8. AccountProjectTest ManageAccount.java Source Code

```
package com.example.example;

/**
 * Title:      Software Testing course
 * Description: The goal of the exercise is implementing a program which
demonstrate a parallel bug.
 * In the exercise we have two accounts.The program enable tranfering
money from one account to the other.Although the functions were defended by
locks (synchronize) there exists an interleaving which we'll experience a
bug.
 * Copyright:   Copyright (c) 2003
 * Company:     Haifa U.
 * @author Maya Maimon
 * @version 1.0
 */

public class ManageAccount extends Thread {
    Account account;
    static Account[] accounts=new Account[10] ;//we may add more later to
increase the parallelism level
    static int num=2;//the number of the accounts
    static int accNum=0;//index to insert the next account
    int i;//the index

    public ManageAccount(String name,double amount) {
        account=new Account (name,amount) ;
        i=accNum;
        accounts[i]=account;
        accNum=(accNum+1)%num;//the next index in a cyclic order
    }

    public void run(){
        account.depsite(300);
        account.transfer(accounts[(i+1)%num],10);
        account.depsite(10);
        account.transfer(accounts[(i+2)%num],10);
        account.withdraw(20);
        account.depsite(10);
        account.transfer(accounts[(i+1)%num],10);
        account.withdraw(100);
    }

    static public void printAllAccounts(){
        for (int j=0;j<num;j++){
            if( ManageAccount.accounts[j]!=null){
                ManageAccount.accounts[j].print();//print it
            }
        }
    }

}

} //end of class com.example.example.ManageAccount
```

9. AccountProjectTest AccountTest.java Source Code

```
package com.example.example;

import org.junit.Test;

import static java.lang.Thread.sleep;
import static org.junit.Assert.*;

public class AccountTest {

    /**
     * Attempt to deposit from an account using two threads
     * RSK causes under-depositing
     *
     * @throws InterruptedException
     */
    @Test
    public void depositRace() throws InterruptedException {
        Account ac = new Account("", 0.0);
        class Incrementer extends Thread{
            Account ac;
            public Incrementer(Account ac){
                this.ac = ac;
            }
            @Override
            public void run(){
                for (int i=0;i<1000000;i++)
                    ac.depsite(1);
            }
        }
        Incrementer i1 = new Incrementer(ac);
        Incrementer i2 = new Incrementer(ac);
        i1.start();
        i2.start();
        sleep(1000);
        assertEquals(2000000.0, ac.amount, 0.0);
    }

    /**
     * Attempt to withdraw from an account using two threads
     * RSK causes under-withdrawing
     *
     * @throws InterruptedException
     */
    @Test
    public void withdrawRace() throws InterruptedException {
        Account ac = new Account("", 0.0);
        class Incrementer extends Thread{
            Account ac;
            public Incrementer(Account ac){
                this.ac = ac;
            }
            @Override
            public void run(){
                for (int i=0;i<1000000;i++)
                    ac.withdraw(1);
            }
        }
        Incrementer i1 = new Incrementer(ac);
        Incrementer i2 = new Incrementer(ac);
```

```

        i1.start();
        i2.start();
        sleep(1000);
        assertEquals(-2000000.0, ac.amount, 0.0);
    }

    /**
     * Attempt to transfer from one account to another using two threads
     * RSK causes under-transferring
     *
     * @throws InterruptedException
     */
    @Test
    public void transferRace() throws InterruptedException {
        Account ac1 = new Account("", 0.0);
        Account ac2 = new Account("", 0.0);
        class Incrementer extends Thread{
            Account ac1;
            Account ac2;
            public Incrementer(Account ac1, Account ac2){
                this.ac1 = ac1;
                this.ac2 = ac2;
            }
            @Override
            public void run(){
                for (int i=0; i<1000000; i++)
                    ac1.transfer(ac2, 1);
            }
        }
        Incrementer i1 = new Incrementer(ac1, ac2);
        Incrementer i2 = new Incrementer(ac1, ac2);
        i1.start();
        i2.start();
        sleep(1000);
        assertEquals(-2000000.0, ac1.amount, 0.0);
    }

    /**
     * Attempt to pay from two accounts into a third account
     * to underfund account 3
     *
     * @throws InterruptedException
     */
    @Test
    public void transferRace2() throws InterruptedException {
        Account ac1 = new Account("", 0.0);
        Account ac2 = new Account("", 0.0);
        Account ac3 = new Account("", 0.0);
        class Incrementer extends Thread{
            Account ac1;
            Account ac2;
            public Incrementer(Account ac1, Account ac2){
                this.ac1 = ac1;
                this.ac2 = ac2;
            }
            @Override
            public void run(){
                for (int i=0; i<1000000; i++)
                    ac1.transfer(ac2, 1);
            }
        }
    }

```

```
    }  
    Incrementer i1 = new Incrementer(ac1,ac3);  
    Incrementer i2 = new Incrementer(ac2,ac3);  
    i1.start();  
    i2.start();  
    sleep(1000);  
    assertEquals(2000000.0,ac3.amount,0.0);  
  }  
}
```


10. DeadlockProjectTest Source Code

```
package com.example.example;

public class Locks {
    public Object lock1;
    public Object lock2;

    public Locks() {
        lock1 = new Object();
        lock2 = new Object();
    }
}

package com.example.example;

class Process1 extends Thread {
    public Locks lock;

    public Process1(Locks lock){
        this.lock = lock;
    }

    public void run() {
        synchronized (lock.lock1) {
            try {
                sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (lock.lock2) {
            }
        }
    }
}

package com.example.example;

class Process2 extends Thread {
    public Locks lock;

    public Process2(Locks lock){
        this.lock = lock;
    }

    public void run() {
        synchronized (lock.lock1) {
            try {
                sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (lock.lock2) {
            }
        }
    }
}
```

11. DeadlockProjectTest DeadlockTest.java Source Code

```
package com.example.example;

import org.junit.Test;

import static java.lang.Thread.sleep;
import static org.junit.Assert.*;

public class DeadlockTest {

    /**
     * Runs the main method of Deadlock on a second thread and
     * tests its execution time to check if a deadlock has occurred
     */
    @Test
    public void checkDeadlocking() throws InterruptedException {
        Locks lock = new Locks();
        Process1 p1 = new Process1(lock);
        Process2 p2 = new Process2(lock);
        p1.start();
        p2.start();
        sleep(1000);
        assertFalse(p1.isAlive());
    }
}
```

12. Evaluation Checklist

ConMuti Application Evaluation Checklist

Project Under Test:

Test Ref.	Acronym	Guide Question	Answer	Details	Fully Tested?	Evaluation Notes
1	MXT	Does the class have wait, await, sleep or join method call expressions with a timeout parameter?	-		-	
2	MSP	Does the class have synchronized blocks?	-		-	
3	ESP	Does the class have multiple, nested synchronized blocks?	-		-	
4	MSF	Does the class have semaphores?	-		-	
5	MXC	Does the class have latches, semaphores or barriers?	-		-	
6	MBR	Does the class have a barrier with an optional runnable parameter?	-		-	
7	RTXC	Does the class have wait, join, sleep, yield, notify or notifyAll method calls?	-		-	

8	RCXC	Does the class have lock, unlock, signal, signalAll, acquire, release or countDown method calls?	-		-	
9	RNA	Does the class have notifyAll method calls?	-		-	
10	RJS	Does the class have join method calls?	-		-	
11	ELPA	Does the class have lock or semaphore locking / permit acquisition?	-		-	
12	EAN	Does the class have a getAndSet method call to an atomic variable?	-		-	
13	ASTK	Does the class have a non-static synchronized method?	-		-	
14	RSTK	Does the class have a static synchronized method?	-		-	
15	RSK	Does the class have a synchronized method?	-		-	
16	RSB	Does the class have a synchronized block?	-		-	

17	RVK	Does the class have a volatile variable?	-		-	
18	RFU	Does the class have a finally keyword wrapping an unlock method call?	-		-	
19	RXO	Does the class have multiple locks, semaphores, latches or cyclicbarriers?	-		-	
20	EELO	Does the class have multiple locks?	-		-	
21	SHCR	Does the class have a synchronized block?	-		-	
22	SKCR	Does the class have a synchronized block?	-		-	
23	EXCR	Does the class have a synchronized block?	-		-	
24	SPCR	Does the class have a synchronized block?	-		-	

13. RaceConditionProjectTest - Evaluation Checklist

ConMuti Application Evaluation Checklist

Project Under Test: RaceCondition.java

Test Ref.	Acronym	Guide Question	Answer	Details	Fully Tested?	Further Evaluation Notes
1	MXT	Does the class have wait, await, sleep or join method call expressions with a timeout parameter?	No		No	
2	MSP	Does the class have synchronized blocks?	No		No	
3	ESP	Does the class have multiple, nested synchronized blocks?	No		No	
4	MSF	Does the class have semaphores?	No		No	
5	MXC	Does the class have latches, semaphores or barriers?	No		No	
6	MBR	Does the class have a barrier with an optional runnable parameter?	No		No	
7	RTXC	Does the class have wait, join, sleep, yield, notify or notifyAll method calls?	No		No	

8	RCXC	Does the class have lock, unlock, signal, signalAll, acquire, release or countDown method calls?	No		No	
9	RNA	Does the class have notifyAll method calls?	No		No	
10	RJS	Does the class have join method calls?	No		No	
11	ELPA	Does the class have lock or semaphore locking / permit acquisition?	No		No	
12	EAN	Does the class have a getAndSet method call to an atomic variable?	No		No	
13	ASTK	Does the class have a non-static synchronized method?	Yes	One non-static, synchronized method at line 14	No	Code should unsuccessfully mutate as count is non-static
14	RSTK	Does the class have a static synchronized method?	No		No	
15	RSK	Does the class have a synchronized method?	Yes	One synchronized method at line 14	Yes	RSK mutation at line 14
16	RSB	Does the class have a synchronized block?	No		No	
17	RVK	Does the class have a volatile variable?	No		No	
18	RFU	Does the class have a finally keyword wrapping an unlock method call?	No		No	

19	R XO	Does the class have multiple locks, semaphores, latches or cyclicbarriers?	No		No	
20	E ELO	Does the class have multiple locks?	No		No	
21	S HCR	Does the class have a synchronized block?	No		No	
22	S KCR	Does the class have a synchronized block?	No		No	
23	E XCR	Does the class have a synchronized block?	No		No	
24	S PCR	Does the class have a synchronized block?	No		No	

14. AccountProjectTest - Evaluation Checklist

ConMuti Application Evaluation Checklist

Class Under Test: Account (project)

Test Ref.	Acronym	Guide Question	Answer	Details	Fully Tested?	Evaluation Notes
1	MXT	Does the class have wait, await, sleep or join method call expressions with a timeout parameter?	No	One join method within Main.java with no timeout parameter	No	
2	MSP	Does the class have synchronized blocks?	Yes	Line 30 in Account.java	Yes	Predicted mutations: MSP (System.out) at line 30 within void transfer method MSP (this) at line 30 within void transfer method All mutations compile
3	ESP	Does the class have multiple, nested synchronized blocks?	No		No	
4	MSF	Does the class have semaphores?	No		No	
5	MXC	Does the class have latches, semaphores or barriers?	No		No	
6	MBR	Does the class have a barrier with an optional runnable parameter?	No		No	
7	RTXC	Does the class have wait, join, sleep, yield, notify or notifyAll method calls?	Yes	Line 36 in Main.java	Yes	Predicted mutations: RTXC mutation where "bank[k].join();" method call removed Mutation compiles

8	RCXC	Does the class have lock, unlock, signal, signalAll, acquire, release or countDown method calls?	No		No	
9	RNA	Does the class have notifyAll method calls?	No		No	
10	RJS	Does the class have join method calls?	Yes	Line 36 in Main.java	Yes	Predicted mutations: RJS mutation where "bank[k].join();" method call becomes "bank[k].sleep(10000);". Mutation compiles
11	ELPA	Does the class have lock or semaphore locking / permit acquisition?	No		No	
12	EAN	Does the class have a getAndSet method call to an atomic variable?	No		No	
13	ASTK	Does the class have a non-static synchronized method?	Yes	Line 13 in Account.java Line 17 in Account.java Line 21 in Account.java Line 34 in Account.java	Yes	Predicted mutations: ASTK mutations at lines 13, 17, 21, 34 on "despite", "withdraw", "transfer" and "print" method declarations. However, only "print" (line 34) shall survive compilation due to non-static contents of other methods.
14	RSTK	Does the class have a static synchronized method?	No		No	
15	RSK	Does the class have a synchronized method?	Yes	Line 13 in Account.java Line 17 in Account.java Line 21 in	Yes	Predicted mutations: RSK mutations at lines 13, 17, 21, 34 on "despite", "withdraw", "transfer" and "print" method declarations. All mutations compile

				Account.java Line 34 in Account.java		
16	RSB	Does the class have a synchronized block?	Yes	Line 30 in Account.java	Yes	Predicted mutations: RSB mutation at line 30 within "transfer" method. Mutation compiles.
17	RVK	Does the class have a volatile variable?	No		No	
18	RFU	Does the class have a finally keyword wrapping an unlock method call?	No		No	
19	RXO	Does the class have multiple locks, semaphores, latches or cyclicbarriers?	No		No	
20	EELO	Does the class have multiple locks?	No		No	
21	SHCR	Does the class have a synchronized block?	Yes	Line 30 in Account.java	No	No mutations as there are no nodes either side of the critical region
22	SKCR	Does the class have a synchronized block?	Yes	Line 30 in Account.java	No	No mutations as a critical region with one node cannot be shrunk (equivalent to RSB mutation operator)
23	EXCR	Does the class have a synchronized block?	Yes	Line 30 in Account.java	No	No mutations as there are no nodes either side of the critical region
24	SPCR	Does the class have a synchronized block?	Yes	Line 30 in Account.java	No	No mutations as a critical region with one node cannot be split

15. DeadlockProjectTest - Evaluation Checklist

ConMut Application Evaluation Checklist

Class Under Test: Deadlock (Process1.java)

Test Ref.	Acronym	Guide Question	Answer	Details	Fully Tested?	Evaluation Notes
1	MXT	Does the class have wait, await, sleep or join method call expressions with a timeout parameter?	Yes	sleep at line 13 (100ms)	Yes	Predicted mutations: Two MXT mutations as sleep is both doubled and halved at line 13
2	MSP	Does the class have synchronized blocks?	Yes	Line 11 Line 17	Yes	Predicted mutations: MSP (System.out) at line 11 within run method MSP (this) at line 17 within run method MSP (System.out) at line 11 within run method MSP (this) at line 17 within run method All mutations compile
3	ESP	Does the class have multiple, nested synchronized blocks?	Yes	Line 11 Line 17	Yes	Predicted mutation ESP at line 11 / line 17 where lock objects are switched Mutation compiles
4	MSF	Does the class have semaphores?	No		No	
5	MXC	Does the class have latches, semaphores or barriers?	No		No	
6	MBR	Does the class have a barrier with an optional runnable parameter?	No		No	

7	RTXC	Does the class have wait, join, sleep, yield, notify or notifyAll method calls?	Yes	sleep at line 13 (100ms)	No	Mutation does not compile as surrounding try statement would be empty (syntax error)
8	RCXC	Does the class have lock, unlock, signal, signalAll, acquire, release or countDown method calls?	No		No	
9	RNA	Does the class have notifyAll method calls?	No		No	
10	RJS	Does the class have join method calls?	No		No	
11	ELPA	Does the class have lock or semaphore locking / permit acquisition?	No		No	
12	EAN	Does the class have a getAndSet method call to an atomic variable?	No		No	
13	ASTK	Does the class have a non-static synchronized method?	No		No	
14	RSTK	Does the class have a static synchronized method?	No		No	
15	RSK	Does the class have a synchronized method?	No		No	

16	RSB	Does the class have a synchronized block?	Yes	Line 11 Line 17	Yes	Predicted mutations: RSB mutations at lines 11 and 17 within run method Mutations compile
17	RVK	Does the class have a volatile variable?	No		No	
18	RFU	Does the class have a finally keyword wrapping an unlock method call?	No		No	
19	R XO	Does the class have multiple locks, semaphores, latches or cyclicbarriers?	No		No	
20	EELO	Does the class have multiple locks?	No		No	
21	SHCR	Does the class have a synchronized block?	Yes	Line 11 Line 17	No	No mutations as critical regions are either empty or encapsulate the full parent critical region
22	SKCR	Does the class have a synchronized block?	Yes	Line 11 Line 17	No	No mutations as critical regions do not have 3 or more nodes
23	EXCR	Does the class have a synchronized block?	Yes	Line 11 Line 17	No	No mutations as both critical regions are at the end of their parent critical region
24	SPCR	Does the class have a synchronized block?	Yes	Line 11 Line 17	Yes	Synchronized block at line 11 can be split between the try/catch statement below and the nested synchronized block