MUTATION TESTING OF CONCURRENT PROGRAMS

by

Mehmet Köse

B.S., Computer Engineering, Ankara University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2016

MUTATION TESTING OF CONCURRENT PROGRAMS

APPROVED BY:

Assoc. Prof. Alper Şen (Thesis Supervisor)	.....
Prof. Can Özturan	.....
Assist. Prof. Barış Aktemur	.....

DATE OF APPROVAL: 26.08.2016

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Alper Şen for keeping me motivated, fellow jury members Prof. Can Özturan and Assist. Prof. Barış Aktemur for all the feedback, my mother Fatma Köse and my father Şükrü Köse for their endless support, dear schoolmates Yavuz Köroğlu, Gökçehan Kara, and Osman Aktepe for all the brainstorming sessions we had, and the most importantly those others which I forgot to mention here for all their help.

ABSTRACT

MUTATION TESTING OF CONCURRENT PROGRAMS

As software testing becomes a more established discipline, the need for quality assurance of software test suites arises. It has been decades since mutation testing was first introduced into the literature for this purpose. In this thesis, we investigate mutation testing of concurrent Java programs.

We have developed a publicly available mutation testing tool, "Mutanticide" [38], for this purpose. Additionally, we propose a new mutation operator, and use it in combination with previously introduced 14 mutation operators [9]. We describe each of these mutation operators and potential bugs they introduce into the system. Using a bytecode instrumentation tool [4], we explain how to discover mutation candidates and create mutants of a program. Experiments on varying size of software show that programs which have test suites focusing on concurrency catch more multi-threaded bugs. Also, our new operator, RNNA, is proven by selective mutation to be efficient and effective as it produces low number of mutants with a high percentage of dead mutants. In addition, RNNA ends up in the sufficient subset of concurrent mutation operators, too. Another mutation operator that we propose in this study, RSR, targets programs making use of the Java library *java.util.concurrent*. This operator is the first step of our work to develop more mutation operators regarding the aforesaid library.

ÖZET

EŞ ZAMANLI PROGRAMLARIN MUTASYON TESTİ

Yazılım testi daha yerleşik bir disiplin haline geldikçe, yazılım testleri için kalite kontrol yapılması ihtiyacı da doğmaktadır. Mutasyon testinin bu amaçla literatüre giriş yapmasının üzerinden onyıllar geçti. Bu tezde, eş zamanlı Java programlarının mutasyon testini inceliyoruz.

Bu amaçla, "Mutanticide" [38] isminde, açık ve özgür bir mutasyon testi aracı geliştirdik. Ek olarak, yeni bir mutasyon operatörü öneriyor, daha önceden var olan 14 mutasyon operatörüyle beraber kullanıyoruz. Bu mutasyon operatörlerinin her birini ve sisteme kattıkları potansiyel hataları tanıtıyoruz. Bir baytkod enstrümantasyon aracı kullanarak [4], mutasyon adaylarının nasıl bulunabileceğini ve programın mutantlarının nasıl üretilebileceğini açıklıyoruz. Değişen boyuttaki yazılımlar üzerinde yapılan deneyler, eş zamanlı programlamaya odaklanan test kümesine sahip programların daha fazla eş zamanlı hata yakaladığını gösteriyor. Ayrıca, yeni operatörümüz RNNA'nın, etkili ve verimli olduğu, ürettiği düşük mutant sayısı ve yüksek ölü mutant yüzdesinden dolayı, seçmeli mutasyon yöntemiyle ispatlanıyor. Ek olarak, RNNA çok iş parçacıklı mutasyon operatörlerinin yeterli alt kümesi içine de giriyor. Bu çalışmada önerdiğimiz diğer bir mutasyon operatörü olan RSR ise *java.util.concurrent* Java kütüphanesini kullanan programları hedefliyor. Bu operatör, bahsi geçen kütüphaneye dair geliştirmeyi planladığımız mutasyon operatörlerinin ilk adımını teşkil ediyor.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xi
LIST OF ACRONYMS/ABBREVIATIONS xii
1. INTRODUCTION 1
1.1. Contributions 3
1.2. Outline 3
2. BACKGROUND 4
2.1. Software Testing and Concurrent Programs 4
2.2. Mutation Testing 5
2.3. Bytecode Instrumentation 7
2.4. Selective Mutation 8
3. RELATED WORK 9
4. MUTATION OPERATORS 13
4.1. Keyword Mutation Operators 13
4.1.1. ASK (Add Synchronized Keyword) 13
4.1.2. ASTK (Add Static Keyword) 15
4.1.3. RSK (Remove Synchronized Keyword) 15
4.1.4. RSTK (Remove Static Keyword) 16
4.1.5. RVK (Remove Volatile Keyword) 16
4.2. Critical Region Mutation Operators 17
4.2.1. EXCR (Expand Critical Region) 17
4.2.2. SHCR (Shift Critical Region) 17
4.2.3. SKCR (Shrink Critical Region) 19
4.2.4. SPCR (Split Critical Region) 21
4.2.5. RSB (Remove Synchronized Block) 22
4.3. Method Call Mutation Operators 22

4.3.1. MXT (Modify method-X Time) . . . 23
4.3.2. RJS (Replace Join with Sleep) . . . 23
4.3.3. RTXC (Remove Thread method-X Call) . . . 23
4.3.4. RNAN (Replace NotifyAll with Notify) . . . 24
4.3.5. RNNA (Replace Notify with NotifyAll) . . . 25
4.3.6. RSR (Remove Semaphore Release) . . . 26
5. MUTATION TESTING TOOL . . . 28
5.1. Class Traverser . . . 30
5.1.1. Keyword Mutation Candidates . . . 30
5.1.2. Critical Region Mutation Candidates . . . 31
5.1.3. Method Call Mutation Candidates . . . 31
5.2. Mutant Creation . . . 32
5.2.1. ASK . . . 35
5.2.2. ASTK . . . 35
5.2.3. RSK . . . 35
5.2.4. RSTK . . . 35
5.2.5. RVK . . . 35
5.2.6. EXCR . . . 36
5.2.7. SHCR . . . 36
5.2.8. SKCR . . . 36
5.2.9. SPCR . . . 36
5.2.10. RSB . . . 37
5.2.11. MXT . . . 37
5.2.12. RJS . . . 37
5.2.13. RTXC . . . 37
5.2.14. RNAN . . . 37
5.2.15. RNNA . . . 38
5.2.16. RNNA . . . 38
6. EXPERIMENTS . . . 39
6.1. Experiment 1 . . . 41
6.2. Experiment 2 . . . 44

6.3. Experiment 3 46
6.4. Experiment 4 50
7. CONCLUSION AND FUTURE WORK 52
7.1. Limitations 52
7.2. Future Work 53
REFERENCES 54
APPENDIX A: SAMPLE PROGRAM TO DEMONSTRATE RNNA 60
APPENDIX B: SAMPLE PROGRAM TO DEMONSTRATE RSR 65

LIST OF FIGURES

Figure 2.1	Mutation Testing Flow Diagram	6
Figure 4.1	Before ASK Mutation	14
Figure 4.2	After ASK Mutation	14
Figure 4.3	Before ASTK Mutation	15
Figure 4.4	After ASTK Mutation	15
Figure 4.5	Before RSK Mutation	15
Figure 4.6	After RSK Mutation	16
Figure 4.7	Before RSTK Mutation	16
Figure 4.8	After RSTK Mutation	16
Figure 4.9	Before RVK Mutation	17
Figure 4.10	After RVK Mutation	17
Figure 4.11	Before EXCR Mutation	18
Figure 4.12	After EXCR Mutation	18
Figure 4.13	Before SHCR Mutation	19
Figure 4.14	After SHCR Mutation	19
Figure 4.15	Before SKCR Mutation	20
Figure 4.16	After SKCR Mutation	20
Figure 4.17	Before SPCR Mutation	21
Figure 4.18	After SPCR Mutation	21
Figure 4.19	Before RSB Mutation	22
Figure 4.20	After RSB Mutation	22
Figure 4.21	Before MXT Mutation	23
Figure 4.22	After MXT Mutation	23
Figure 4.23	Before RJS Mutation	24
Figure 4.24	After RJS Mutation	24
Figure 4.25	Before RTXC Mutation	24

Figure 4.26 After RTXC Mutation 25
Figure 4.27 Before RNAN Mutation 25
Figure 4.28 After RNAN Mutation 25
Figure 4.29 Before RNNA Mutation 26
Figure 4.30 After RNNA Mutation 26
Figure 4.31 Before RSR Mutation 27
Figure 4.32 After RSR Mutation 27
Figure 5.1 Mutanticide Flow Diagram 29
Figure 5.2 Keyword Mutation Candidates Algorithm 30
Figure 5.3 A method with access modifiers *public* and *synchronized* . . 31
Figure 5.4 A field with access modifiers *private* and *volatile* 31
Figure 5.5 Critical Region Mutation Candidates Algorithm 32
Figure 5.6 Method Call Mutation Candidates Algorithm 33
Figure 5.7 Create Mutants Algorithm 34
Figure 6.1 Run Mutation Testing Algorithm 40
Figure A.1 Producer Consumer 60
Figure A.2 Producer 61
Figure A.3 Consumer 62
Figure A.4 Bounded Buffer 63
Figure A.5 Counting Semaphore 64
Figure A.6 Binary Semaphore 64
Figure B.1 Driver Program 65
Figure B.2 Sample Thread 66

LIST OF TABLES

Table 4.1	Concurrent Mutation Operators	14
Table 6.1	Benchmark Programs for Experiment 1	41
Table 6.2	Results for Experiment 1	42
Table 6.3	Mutant distribution for Experiment 1	43
Table 6.4	Selective Mutation for Experiment 1	44
Table 6.5	Benchmark Programs for Experiment 2	45
Table 6.6	Results for Experiment 2	45
Table 6.8	Selective Mutation for Experiment 2	46
Table 6.7	Mutant distribution for Experiment 2	46
Table 6.9	Benchmark Programs for Experiment 3	47
Table 6.10	Results for Experiment 3	47
Table 6.11	Mutant distribution for Experiment 3	48
Table 6.12	2-selective Mutation	49
Table 6.13	Sufficient set of mutation operators	50
Table 6.14	Benchmark Programs for Experiment 4	50
Table 6.15	Results for Experiment 4	50
Table 6.16	Mutant distribution for Experiment 4	51

LIST OF ACRONYMS/ABBREVIATIONS

CPU	Central Processing/Processor Unit
CR	Critical Region
JVM	Java Virtual Machine
KW	Keyword
LoC	Lines of Code
MC	Method Call
MSG	Mutant Schemata Generation
SB	Synchronized Block
SUT	Software Under Test

1. INTRODUCTION

Physical limitations on the maximum frequence of a CPU core forced the CPU designers to take on a different approach for the last decade. Instead of trying and producing single-core heavy-weight CPUs, the design trend has evolved into multi-core architecture with multiple light-weight CPU cores. In order to fully utilize the power of this architecture, hardware optimizations fall short. Thus, software engineers need to develop multi-threaded programs for multi-core systems. Since this is a quite different approach to software development, it introduces new bugs into software systems. A multi-threaded software can take on a different schedule on each run, therefore data races now become a major and a more frequent issue, as well as deadlocks. These new types of bugs require a different testing approach.

In its infancy years, software testing was seen no different than debugging. Then was born the idea that testing was to show the absence of errors, which was shortly replaced with testing to show the presence of errors. As software engineering excelled and began to evolve into an advanced discipline, philosophy of testing also improved. Biggest and most successful software companies today accept testing as a mindset that enables developers and testers to produce higher quality software. Mutation testing is fully consistent with this approach. In the process of mutation testing, mutants of the System Under Test (SUT) are created and the test suite is run on each mutant to see whether they produce the same output or not. If the test case run on a mutant produces a different output than the same test case run on the original SUT, we say that the mutant is *killed* by that test case. Otherwise, the mutant is said to be *alive*. While sequential programs have one possible outcome, concurrent programs may have multiple. A mutant of a concurrent program is killed if its output is different than that of all possible outputs of the SUT [47].

A test suite's code coverage reaching 100% proves the quality of neither the software nor the test suite. With mutation testing, however, engineers can be sure what kind of errors their test suite catches, and in effect, their software is free of.

There is also empirical evidence that mutation testing is superior to most techniques: Walsh [52] notes that "mutation testing is more powerful than statement or branch coverage". Furthermore, Frankl *et al.* [22] states "mutation testing is superior to data flow coverage critera".

Mutation testing has its drawbacks, too. Its leading disadvantage is high amount of time required for mutant generation and execution. A SUT with a few hundreds of lines of code (LoC) can generate hundreds, or maybe thousands in extreme cases, of mutants. Compiling these mutated versions and running each test case against each mutant takes a serious amount of time, thereof preventing mutation testing to be accepted as a common testing technique in the industry. A significant number of cost reduction techniques have been proposed and successfully applied so far.

Usaola and Mateo [51] divides the cost reduction techniques into three, based on the phases they are applied in: *mutant generation*, *test case generation and execution*, and *result analysis*. Selective mutation is the major mutant count reduction technique in use. The idea is using only a subset of mutation operators without sacrificing effectiveness much. It also has been proven to produce fewer equivalent mutants, since it produces lower number of mutants on the whole. More details can be found in Section 2.4.

The most common way of reducing mutant execution time is *partial mutation testing*, where each mutant is run against test cases until it is *killed*. The dual of this approach is *full mutation testing* in which a mutant is run against all test cases. Another technique in this category is *weak mutation* [26], whose opposite is *strong mutation*. In strong mutation, mutants are killed by comparing the terminal program state, whereas in weak mutation, a mutant is killed as soon as a different program state is run across during execution.

Though mutation testing is not a technique unknown prior to concurrent software, its application on multi-threaded programs remains rather unexplored. Our aim in this thesis is to develop a high-end mutation testing tool for concurrent Java programs.

1.1. Contributions

- We developed a mutation testing tool, Mutanticide, for concurrent Java programs which is the first one to be publicly available, to the best of our knowledge.
- We proposed a new concurrent mutation operator, RNNA (Replace Notify with Notify All). We observed that mutants created by this operator caused data races in some programs, which is a major bug in concurrent programs.
- We proposed another concurrent mutation operator, RSR (Remove Semaphore Release), that targets the Java library *java.util.concurrent*.
- We ran experiments to analyze how sufficient test suites of various benchmarks are, and also inspected the necessity of our newly proposed mutation operator using selective mutation.

1.2. Outline

In Chapter 2, we give a background on software testing and mutation testing. Furthermore, we explain the process of mutation testing, and express its historical and theoretical grounds. Chapter 3 introduces the related work on mutation testing, especially for Java programs and concurrent software. In Chapter 4, we explain the mutation operators for concurrent software that we used in this work. Chapter 5 introduces the mutation testing tool we developed, Mutanticide. We present the experiments we ran in Chapter 6, and discuss their outcome. Lastly, we arrive at a conclusion and state our future work in Chapter 7.

2. BACKGROUND

In this chapter, we first describe mutation testing with its historical and theoretical background, explain the process of mutation testing. Concepts such as mutation operator and mutation score are clarified. Then, we discuss the concurrency mechanisms and the bugs they introduce. Lastly, we describe two mutation testing cost reduction techniques, bytecode instrumentation and selective mutation.

2.1. Software Testing and Concurrent Programs

As the world becomes more digital, we get surrounded by more software. We have witnessed software failures that cost millions of dollars, caused security breaches, or even loss of human lives. Most of these disasters were due to absence or insufficiency of testing which is a crucial part of the software development process. Therefore, today's leading software companies put testing in the center of their working discipline. A former Google executive explains their view as "*If you are an engineer, you are a tester. If you are an engineer with the word test in your title, then you are an enabler of good testing for those other engineers who do not*" [53].

With the improvements in computer technology, hardware systems evolve, too. Multi-core and multi-processor architectures have been the standard even for personal computers and smart phones. Although these state of the art microchips have hardware optimizations to work efficiently, we need to develop multi-threaded software to fully utilize their potential. However, this new paradigm of concurrency introduces new bugs to the systems. Unlike sequential programs which run one instruction at a time, concurrent programs consist of multiple threads that run simultaneously. When these threads lack their coordination with each other, programs may produce incorrect results due to data race, or end up in a deadlock. These distinctive bugs require proper tests that specialize in the same manner.

Modern programming languages such as Java provide concurrency mechanisms that enable programmers write better and more reliable concurrent programs. Unfortunately, this does not make the programs hazard-free. For instance, a critical region in Java can be created by writing a synchronized block, which enables atomicity and exclusive execution of the code segment inside it. However, poorly deciding which statements to include or exclude in this block, or not creating one where it is needed, will most likely cause a failure. Another example of a common fault is calling the wrong concurrency method, or setting a faulty parameter for the method invocation. More examples on this matter can be seen in Chapter 4.

2.2. Mutation Testing

Mutation testing is the concept of injecting simple faults into the SUT and run the test suite over these defective copies, which are called *mutants*, to see whether the tests catch these faults in the program. *Mutation operators* are the rules that define what kind of faults to be injected into the SUT. These faults should resemble the ones that programmers commit in practice for a more accurate estimation of a test suite's quality.

Mutation testing dates back to 1978 and relies on two hypotheses [17]: Competent Programmer Hypothesis and Coupling Effect. The former states that programmers, on the whole, create programs that are *close* to being perfect. In effect, faults they commit are simple. The latter corresponds to the idea that simple faults are coupled with complex faults. In other words, if a test suite covers simple faults, it will ensure to have covered most of the complex faults.

Due to the hypotheses mentioned above, mutations are simple faults that are injected into a program. Mutation operators, first proposed and implemented in [33], are defined and used to produce mutant copies of the SUT. Later on, the test suite of the SUT is run against these mutant versions. If a mutant is observed to present different behavior than that of the original program, it is said to have been *killed*. Otherwise, it remains *alive*. This work-flow of mutation testing is represented in Figure 2.1.

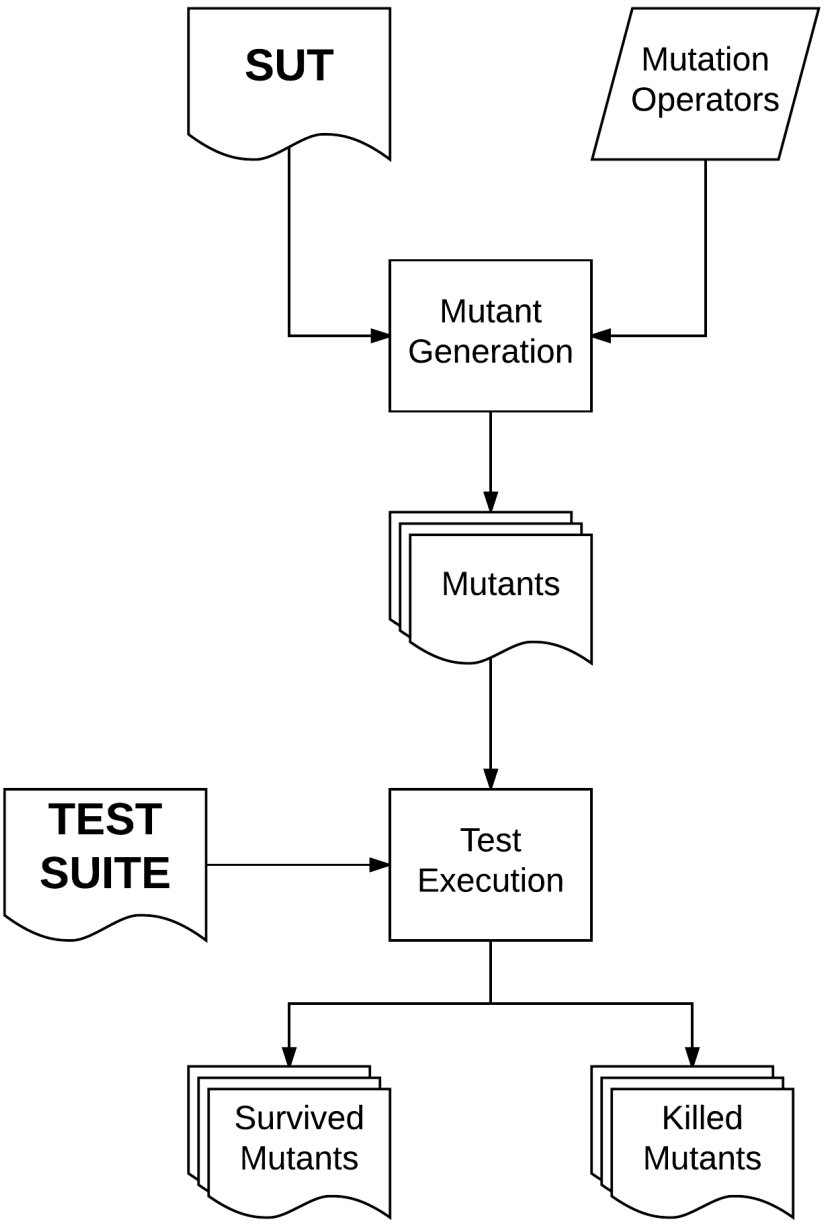

Figure 2.1: Mutation Testing Flow Diagram.

Upon completion of the testing process, the test suite is refined to kill the surviving mutants. However, there are some mutants that can never be killed, namely *equivalent mutants*. These are syntactically different than, but semantically identical to the SUT. Hence they always produce the same output with the original program. Presence of equivalent mutants affect the result analysis cost seriously. Automatically detecting them is impossible since program equivalence is undecidable [11]. Thus, this phase requires human effort, leading to higher costs. The research addressing this problem is mostly on reducing equivalent mutant count, rather than easing equivalence detection.

A test suite's quality is determined by its *mutation score*. This value is given by the ratio $\frac{\#Mutants_killed}{\#All_mutants}$. The higher the ratio, the better the test suite is. If mutation score is thought to be not high enough, test suite is revised and run again until a satisfactory mutation score is reached.

2.3. Bytecode Instrumentation

Bytecode is an intermediate language that can be read by an interpreter. Unlike source codes that are human-readable, bytecodes consist of numeric codes, constants, opcodes, and predefined instructions [54]. On the other hand, reading and manipulating bytecode is easier in comparison with the machine code, which consists of nothing but binary numbers. Bytecode instrumentation is the act of adding/removing/changing instructions, or elements of instructions, in the bytecode of a program.

In the context of the Java programming language, bytecode is an intermediate language which is derived from Java source code and interpreted by the Java Virtual Machine (JVM). Using bytecode instrumentation to scan the SUT and to create mutants eliminates the time needed for compilation of each mutant. It also relieves us from the need for source code of the SUT. Modifying the bytecode is no less safer than working on the source code. Since all lines of statements in the code contain a line number instruction in their bytecode, we can pinpoint the exact location of our mutation candidates, and furthermore, safely discriminate between different mutants

of the same mutation type.

There are a number of bytecode instrumentation libraries for Java, which can be exemplified as BCEL [7], SERP [48], and ASM [4]. The reason we chose ASM is its overwhelming performance over other bytecode instrumentation libraries that provide the same functionality [10].

2.4. Selective Mutation

The most important drawback of mutation testing is the time required to generate and execute mutants being high. One of the reasons for this long execution duration is high mutant count. Selective mutation is among the major approaches to overcome this problem. The operators that generate the most mutants are called *dominant operators*. Selective mutation is a technique to remove the two most dominant operators from the operator pool. It was originally proposed by Mathur [37].

Offutt *et al.* [41] took Mathur's approach one step further and defined *n-selective mutation* for $n \in \{2, 4, 6\}$, where 2, 4, and 6 most dominant mutation operators are discarded, respectively. In terms of selective mutation, an operator becomes better as its number of mutants decreases and its percentage of dead mutants increases. In this study, we apply 2-selective mutation to analyze the efficiency of mutation operators we used.

In addition to these approaches, a sufficient subset of mutation operators can be selected to decrease the number of operators and achieve time improvements [6, 25, 39]. Mutants created by subset S and killed by the test suite make up a fraction F of all mutants. The closer F is to the mutation score, the better it is. Also, as the time it takes to create and test the mutants of S decreases, it becomes more cost efficient. Sufficient set problem is finding a subset of operators which is efficient and sufficient enough.

3. RELATED WORK

There have been a large number of mutation tools for various systems which span from finite state machines to web services. Jia and Harman [29] inform that highest number of mutation tools were developed for Java, which corresponds to 22% of all tools. Interestingly, Fortran comes in second place with 19% but that can be elaborated by the fact that first mutation operators were developed for Fortran [33]. The most notable mutation tools developed for the Java programming language can be listed as follows:

muJava [34] is one of the earliest Java mutation tools. It uses bytecode instrumentation (via BCEL [2]) and mutant schemata generation (MSG) techniques. It has a large set of mutation operators which the authors divide into two categories: *behavioral* and *structural*, the former is the set of operators that change the behavior of the SUT while the latter is the set of operators that change the structure. They utilize MSG method for behavioral mutants and direct bytecode transformation for structural mutants, to increase the efficiency of mutant generation process.

Javalanche [49] is among the best mutation tools developed up to day in terms of both efficiency and effectiveness. It makes use of selective mutation, MSG, coverage data, and bytecode manipulation (via ASM [4]), for efficient processing. Additionally, it can execute mutants in parallel to further increase efficiency.

Judy [35] follows a whole different design path. It is an implementation of the FAMTA Light (Fast Aspect-oriented Mutation Testing Algorithm) approach developed in Java with AspectJ [5] extensions. It basically takes advantage of the pointcut and advice mechanisms. The most important limitation of this tool is the maximum number of mutations that could be applied to a single Java class within an iteration must be kept low (optimally between 50 and 100), otherwise *out of memory exception* is thrown.

AJMutator [15] is a tool for the mutation analysis of AspectJ pointcut descriptors (PCDs). It produces mutants of AspectJ aspects by inserting faults in the PCDs.

Jumble's [27] most noticeable feature is its use of a separate Java Virtual Machine (JVM) so that if a failure occurs during the execution of one mutation, then the child JVM can be restarted and the testing process goes on smoothly. It uses BCEL [2] to manipulate bytecode and uses BCEL's customizable class loader, which Irvine *et al.* [27] state to cause problems with systems that implement their own class loaders.

Bacterio [36] is a very advanced mutation tool which employs a high number of efficiency techniques. Apart from well-known methods such as selective mutation, weak mutation, etc. it uses *mutant sampling* (selecting a subset of mutants randomly), and *flexible mutation* (a technique between weak and strong mutation). Though this tool is GUI-based, it also supports execution scripts to enable automation.

Mugamma [32] is based on the Gamma [43] framework that has been developed to monitor software after deployment. MuGamma collects data from actual users of the software and checks which one of these would have killed mutants. The executions' states which kill a mutant are recorded and later used to create test cases in regression testing when the software is changed.

Major [31] employs a completely different technique. Instead of focusing on source code or bytecode mutation like all other tools, Just comes up with the idea of *compiler-integrated mutation*, which transforms the abstract syntax tree. Empirical results are promising, but only a few mutation operators have been implemented so far. Moreover, it fails to mutate some classes due to their size, Delahaye and Bousquet [14] reports.

Testooj [21] is a tool specifically designed to test component-based systems. Thus, its focus is on integration testing, unlike most mutation tools which focus on unit testing.

To the best of our knowledge, PIT [12] is the state-of-the-art mutation testing tool. It achieves the highest mutation score in almost all experiments compared to other major tools [14].

All these tools implement traditional and object oriented mutation operators. To the best of our knowledge, only two tools implement concurrent Java mutation operators, ExMan [9] and Comutation [25], none of which is publicly available. Additionally, a framework named MutMut [24] was developed for efficient state space exploration of multi-threaded programs, which can be used for concurrent mutation testing.

ExMan is a GUI-based mutation testing tool and can be applied to Fortran, C and Java programs. In order to generate mutants, Bradbury *et al.* use existing mutant generators such as Andrews and Zhang C mutant generator tool [1], along with several mutant generators they wrote using a source transformation language, TXL [13], which is similar to BNF. They state to have combined the features of Mothra [18], Proteum [16], and MuJava, presenting a multi-platform mutation testing tool. In favor of being customizable, they sacrifice the efficiency and effectiveness of a language-dedicated and standardized tool.

Gligoric and his colleagues had developed MutMut. In this work, they built their framework on top of Java Path Finder [28] and made several techniques available for concurrent program exploration. They implemented some concurrent mutation operators in Javalanche and ran experiments. Results show that MutMut provides a speed up ranging from 22% up to 77%.

In addition to MutMut, Gligoric *et al.* [25] developed a mutation tool for concurrent Java programs, named Comutation, and proposed three new concurrent mutation operators. They also implement selective mutation in their tool and report the achievements (in terms of both mutation score and time savings) of different subsets of operators. One of the new operators, *WTIS* (replace while with if inside synchronized), ends up in sufficient sets of operators in the experiments. Unfortunately, they utilize neither MSG nor bytecode instrumentation.

Selective mutation has been applied to different type of programming languages before it has been used for Java. Offutt *et al.* [42] observed four-fold or more reduction in number of mutants while achieving almost the same coverage for Fortran programs. Barbosa et. al [6] applied it on C programs using Proteum [16] and reported to have over 65% of cost reduction.

4. MUTATION OPERATORS

In this chapter we introduce the mutation operators we used. A mutation operator is a rule that define a kind of fault to inject into the SUT. We selected 14 concurrent mutation operators from the literature [8], and proposed two more, RNNA and RSR. All of these 16 mutation operators can be seen in Table 4.1. We divide the operators into three categories based on the modifications they do on a class or a method. These categories are, keyword (KW) mutation, critical region (CR) mutation, and method call (MC) mutation. KW mutation operators alter the access modifier of a class's method or field. CR mutation operators make changes on a synchronized block of a method. They change the size or content of the synchronized block, or totally remove it. Lastly, MC mutation operators make changes on a method invocation statement. This change can be altering the method called, modifying one of its parameters, or removing the invocation statement from the program. Next, we are going to describe these mutation operators in detail.

4.1. Keyword Mutation Operators

This group of mutation operators add or remove concurrency-related keywords from class components. These keywords are the following: synchronized, static, and volatile.

4.1.1. ASK (Add Synchronized Keyword)

This operator adds the "synchronized" keyword to a non-synchronized method. A synchronized method's body acts like a critical region in Java, so that no multiple threads can execute this method concurrently. An example of a sample program before and after this mutation is applied are shown in Figure 4.1 and in Figure 4.2, respectively,

Table 4.1: Concurrent Mutation Operators

KW Mutation	**ASK** (Add Synchronized Keyword)
	ASTK (Add Static Keyword)
	RSK (Remove Synchronized Keyword)
	RSTK (Remove Static Keyword)
	RVK (Remove Volatile Keyword)
CR Mutation	**EXCR** (Expand Critical Region)
	SHCR (Shift Critical Region)
	SKCR (Shrink Critical Region)
	SPCR (Split Critical Region)
	RSB (Remove Synchronized Block)
MC Mutation	**MXT** (Modify Method-X Time)
	RJS (Replace Join with Sleep)
	RTXC (Remove Thread Method-X Call)
	RNAN (Replace NotifyAll with Notify)
	RNNA (Replace Notify with NotifyAll)
	RSR (Remove Semaphore Release)

```
1 public class SUT {
2         public void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.1: Before ASK Mutation.

```
1 public class SUT {
2         public synchronized void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.2: After ASK Mutation.

4.1.2. ASTK (Add Static Keyword)

The operator ASTK adds the "static" keyword to a non-static method. A static method does not belong to an object, but to a class, which makes it possible to call this class without creating an instance of that class. Figure 4.3 shows a sample program before this mutation is applied, and Figure 4.4 shows the same program after the mutation is applied.

```
1 public class SUT {
2         public void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.3: Before ASTK Mutation.

```
1 public class SUT {
2         public static void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.4: After ASTK Mutation.

4.1.3. RSK (Remove Synchronized Keyword)

RSK operator removes the "synchronized" keyword from a synchronized method. A synchronized method's body acts like a critical region in Java. Removing this keyword will make it possible to make multiple concurrent accesses to the method body. Figure 4.5 shows a sample SUT, and Figure 4.6 shows its RSK mutant.

```
1 public class SUT {
2         public synchronized void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.5: Before RSK Mutation.

```
1 public class SUT {
2         public void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.6: After RSK Mutation.

4.1.4. RSTK (Remove Static Keyword)

RSTK operator removes the "static" keyword from a static method. This results in each instance having and executing a separate copy of the method. if a static method is modifying a static variable inside its body, making it non-static creates a race condition for that variable. A sample program and its RSTK mutant can be seen in Figure 4.7 and in Figure 4.8, respectively.

```
1 public class SUT {
2         public static void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.7: Before RSTK Mutation.

```
1 public class SUT {
2         public void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 4.8: After RSTK Mutation.

4.1.5. RVK (Remove Volatile Keyword)

RVK operator removes the "volatile" keyword from a volatile field of the class. A volatile keyword makes concurrent accesses to the respective variable impossible even though they are not enclosed in synchronized blocks. Removing this keyword from a field will make it an ordinary variable of the class. Figure 4.9 shows an example

program and Figure 4.10 shows its RVK mutant.

```
1 public class SUT {
2     private volatile int x;
3 }
```

Figure 4.9: Before RVK Mutation.

```
1 public class SUT {
2     private int x;
3 }
```

Figure 4.10: After RVK Mutation.

4.2. Critical Region Mutation Operators

We have five operators to modify the size of a critical region. Critical regions in Java are defined by synchronized blocks. Only one thread of an object can execute inside the synchronized block at the same time.

4.2.1. EXCR (Expand Critical Region)

This operator expands a critical region, making it include larger code segments. Statements previously not atomic are now executed together. Other threads waiting on the critical region lock will have to wait more, unnecessarily. Expansion of a critical region can be made in three different ways: upwards, downwards, or both sides. An example program can be seen in Figure 4.11. Since this SUT has statements both before and after the critical region, we can expand the critical region in both directions, ending up with the mutant shown in Figure 4.12.

4.2.2. SHCR (Shift Critical Region)

The operator SHCR shifts the start and end of a critical region so that some new statements are included inside the synchronized block and some previously included statements are excluded. Similar to expanding a critical region, shifting it can be done

```
1  public class SUT {
2      public void myMethod(){
3          // Statement 1
4          synchronized(this){
5              // Statement 2
6              // Statement 3
7          }
8          // Statement 4
9          }
10 }
```

Figure 4.11: Before EXCR Mutation.

```
1  public class SUT {
2      public void myMethod(){
3          synchronized(this){
4              // Statement 1
5              // Statement 2
6              // Statement 3
7              // Statement 4
8          }
9      }
10 }
```

Figure 4.12: After EXCR Mutation.

in multiple ways. An example program is shown in Figure 4.13 with a critical region that has a statement before it, but no statements after it. Hence, we can shift the synchronized block upwards and have the mutant shown in Figure 4.14.

```
1 public class SUT {
2     public void myMethod(){
3         // Statement 1
4         synchronized(this){
5             // Statement 2
6             // Statement 3
7         }
8     }
9 }
```

Figure 4.13: Before SHCR Mutation.

```
1 public class SUT {
2     public void myMethod(){
3         synchronized(this){
4             // Statement 1
5             // Statement 2
6         }
7         // Statement 3
8     }
9 }
```

Figure 4.14: After SHCR Mutation.

4.2.3. SKCR (Shrink Critical Region)

SKCR operator shrinks a critical region, moving some statements out of this block. Thus, those statements that were previously part of atomicity will now be executed in a regular way. For a synchronized block to be shrunk, it must contain at least 2 lines of statements. Shrinking can be done from both sides if the synchronized block contains 3 or more statements, like the example given in Figure 4.15. The SKCR mutant of the sample program can be seen in Figure 4.16.

```
1 public class SUT {
2     public void myMethod(){
3         synchronized(this){
4             // Statement 1
5             // Statement 2
6             // Statement 3
7         }
8     }
9 }
```

Figure 4.15: Before SKCR Mutation.

```
1 public class SUT {
2     public void myMethod(){
3         // Statement 1
4         synchronized(this){
5             // Statement 2
6         }
7         // Statement 3
8     }
9 }
```

Figure 4.16: After SKCR Mutation.

4.2.4. SPCR (Split Critical Region)

This operator splits a critical region into two critical regions. Although all previously synchronized statements are still synchronized, each half can be accessed concurrently with each other now. Synchronized blocks containing a single statement cannot be split. In the case of a synchronized block having a high number of statements inside it, multiple ways to shrink are possible. Figure 4.17 shows a SUT with a 4-statement critical region. We divide the number of statements by two, and split the synchronized block accordingly, as seen in Figure 4.18.

```
1  public class SUT {
2      public void myMethod(){
3          synchronized(this){
4              // Statement 1
5              // Statement 2
6              // Statement 3
7              // Statement 4
8          }
9      }
10 }
```

Figure 4.17: Before SPCR Mutation.

```
1  public class SUT {
2      public void myMethod(){
3          synchronized(this){
4              // Statement 1
5              // Statement 2
6          }
7
8          synchronized(this){
9              // Statement 3
10             // Statement 4
11         }
12     }
13 }
```

Figure 4.18: After SPCR Mutation.

4.2.5. RSB (Remove Synchronized Block)

RSB operator completely removes the synchronized block. The atomicity of the statements inside the critical region is totally lost after this operation. A sample program with a critical region and its RSB mutant are shown in Figure 4.19 and in Figure 4.20, respectively.

```
1  public class SUT {
2      public void myMethod(){
3          // Statement 1
4          synchronized(this){
5              // Statement 2
6              // Statement 3
7          }
8          // Statement 4
9      }
10 }
```

Figure 4.19: Before RSB Mutation.

```
1  public class SUT {
2      public void myMethod(){
3          // Statement 1
4          // Statement 2
5          // Statement 3
6          // Statement 4
7      }
8  }
```

Figure 4.20: After RSB Mutation.

4.3. Method Call Mutation Operators

Additional to keywords and critical regions, Java also has some methods to provide concurrency. Modifying calls to these methods may change the behavior of the SUT.

4.3.1. MXT (Modify method-X Time)

Some concurrency mechanisms take time arguments, such as *sleep, wait*, etc. Modifying this time parameter may change program behavior, altering the output, or even causing a deadlock or data race. Figure 4.21 shows a sample program with a *sleep* method whose time parameter is 500 milliseconds. MXT mutant operator alters the time parameter by multiplying it by 2, as seen in Figure 4.22. Alternative ways to change the parameter are possible, such as dividing it by 2, or multiplying by 3.

```
1 public class SUT {
2         public void myMethod(){
3             Thread.sleep(500);
4         }
5 }
```

Figure 4.21: Before MXT Mutation.

```
1 public class SUT {
2         public void myMethod(){
3             Thread.sleep(1000);
4         }
5 }
```

Figure 4.22: After MXT Mutation.

4.3.2. RJS (Replace Join with Sleep)

The join method makes the calling thread suspend and wait for completion of the called thread. The sleep method, on the other hand, suspends the calling thread for a specified amount of time, regardless of other threads' behavior. Replacing join with sleep has the potential to cause a deadlock or data race in certain scenarios. A sample program and its RJS mutant are shown in Figure 4.23 and in Figure 4.24, respectively.

4.3.3. RTXC (Remove Thread method-X Call)

There are various methods that provide concurrency in Java. Removing a method invocation of these, may affect the program behavior dramatically. These methods can

```
1 public class SUT {
2     public void myMethod(){
3         // Some statements
4         thread1.join();
5     }
6 }
```

Figure 4.23: Before RJS Mutation.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             thread1.sleep();
5         }
6 }
```

Figure 4.24: After RJS Mutation.

be listed as *wait, sleep, notify, notifyAll, yield*, and *join*. In Figure 4.25, a sample program with a *join* method invocation is shown. This program is mutated by the RTXC mutation operator as in Figure 4.26.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             thread1.join();
5         }
6 }
```

Figure 4.25: Before RTXC Mutation.

4.3.4. RNAN (Replace NotifyAll with Notify)

notifyAll method makes all waiting threads go on their execution, whereas the method *notify* lets only one of the waiting threads to do this. Replacing *notifyAll* with *notify* may cause infinite waiting time for the other threads and prevent program completion. An example program with a call to *notifyAll* and its RNAN mutant are shown in Figure 4.27 and in Figure 4.28, respectively.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4         }
5 }
```

Figure 4.26: After RTXC Mutation.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             notifyAll();
5         }
6 }
```

Figure 4.27: Before RNAN Mutation.

4.3.5. RNNA (Replace Notify with NotifyAll)

The method *notify* awakes one of the waiting threads and makes it continue execution. *notifyAll* method, on the other hand, causes all waiting threads to wake up. Calling *notifyAll* where *notify* should have been called may cause a deadlock or data race. Figure 4.29 shows an example program which is mutated as shown in Figure 4.30.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             notify();
5         }
6 }
```

Figure 4.28: After RNAN Mutation.

An example program to demonstrate the possible effects of this mutation is included in Appendix A. This program is a variation of the producer-consumer problem with 1 producer and 5 consumers. The producer produces items which are the constant value 1, and places them in the buffer until it is full. The consumers consume an item

from the buffer until it is empty. A consumer notifies a producer after it consumed an item, and a producer notifies a consumer after it produces one. The program goes on execution until all consumers have consumed 5 items each. We also keep a static variable for the consumers such that, whenever a consumer consumes an item, it adds up its value to this static variable. At the end of execution, this static variable is expected to have the value 25 since all 5 consumers will have consumed the value 1 for 5 times. When the call to *notify* on line 17 of Figure A.5 is replaced with a call to *notifyAll*, however, we end up with a different result, like 11 instead of 25.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             notify();
5         }
6 }
```

Figure 4.29: Before RNNA Mutation.

```
1 public class SUT {
2         public void myMethod(){
3             // Some statements
4             notifyAll();
5         }
6 }
```

Figure 4.30: After RNNA Mutation.

4.3.6. RSR (Remove Semaphore Release)

The Java library *java.util.concurrent* has a built-in counting semaphore structure which has its own predefined *acquire* and *release* methods. When a thread calls the *acquire* method, it enters a spin-loop until it achieves the lock of the semaphore. When this thread is done with its job, it calls the *release* method so that another thread now can now obtain the lock. Removing the call to *release* will cause a thread with the lock keep it forever and keep other threads in an infinite wait. Figure 4.31 and Figure 4.32 show a program before and after this fault is injected. We will be proposing more mutation operators focusing this library in our future work.

A sample program which has a mutation candidate for RSR is shown in Appendix B. When the call to *release* on line 20 of Figure B.2 is removed, no more threads than the first 2 can achieve the lock since the counting semaphore is defined with the limit 2.

```
1  import java.util.concurrent.Semaphore;
2
3  class MyThread extends Thread {
4     public void run() {
5        try {
6           semaphore.acquire();
7           // Some statements
8        } catch (InterruptedException ie) {
9        } finally {
10          semaphore.release();
11       }
12    }
13 }
```

Figure 4.31: Before RSR Mutation.

```
1  import java.util.concurrent.Semaphore;
2
3  class MyThread extends Thread {
4     public void run() {
5        try {
6           semaphore.acquire();
7           // Some statements
8        } catch (InterruptedException ie) {
9        } finally {
10
11       }
12    }
13 }
```

Figure 4.32: After RSR Mutation.

5. MUTATION TESTING TOOL

We developed a mutation testing tool called "Mutanticide" [38], which is publicly available. To scan classes and generate their mutants, we made use of a Java bytecode instrumentation tool, ASM [4], which has been empirically proven to be the most efficient of all Java bytecode manipulation tools to the best of our knowledge [10].

Our tool takes a class as an argument and creates its mutants based on the operators described in Chapter 4. Each mutant contains one error to satisfy single mutation. Before creating mutants, the class must be traversed in order to specify mutation candidates. We have a class called "*ClassTraverser*" that scans the given class's fields, method declarations and definitions in the bytecode level. After this information is collected, we create actual mutants. Figure 5.1 shows mutation testing work-flow with Mutanticide.

ASM [4] has various packages for bytecode instrumentation, one of which is the tree package. The tree package reads and writes a class and its components as if it was a tree data structure. A class can be loaded into the memory and written back to disk using the class *ClassNode*, which is the root node of the tree. Methods and fields of a class can be loaded and written by *MethodNode* and *FieldNode* classes, respectively. After a class's method is loaded, its instructions must be visited to add them to the instruction list of the method. When *visitEnd* method of the class *MethodNode* is called, all instructions of a method will be in its corresponding instruction list, which is an instance of *InsnList*. There are 16 types of instructions and *InsnList* can contain. For instance, a new line is a *LineNumberNode*, a method invocation is a *MethodInsnNode*, and a jump instruction is a *JumpInsnNode*.

In Section 5.1, we explain how *ClassTraverser* scans a class and identifies mutation candidates. Then, in Section 5.2, we express how mutants are created by giving brief explanation for each type.

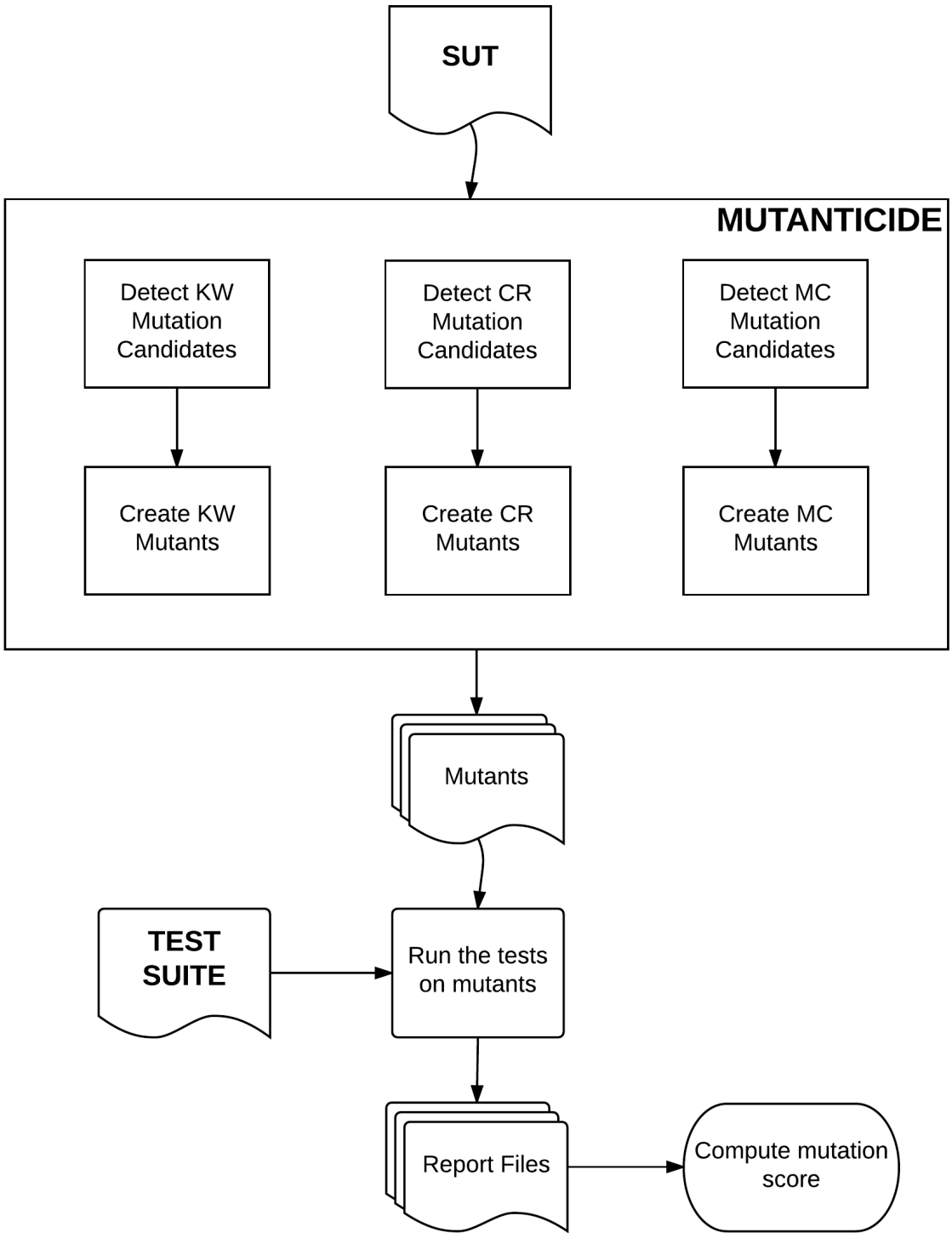

Figure 5.1: Mutanticide Flow Diagram.

5.1. Class Traverser

5.1.1. Keyword Mutation Candidates

Keyword mutations are related to the class's methods and fields, whereas other mutations require to inspect the bytecode of each method instruction by instruction. Therefore, we begin by traversing the class's "metadata" and inspect the access modifiers of its fields and methods. We store the names of non-volatile fields, non-static and static methods, non-synchronized and synchronized methods to their respective lists. These mutation candidates will later transform into RVK, ASTK, RSTK, ASK, RSK mutants. The algorithm to detect KW mutation candidates can be seen in Figure 5.2

```
Input: Java Class C
Output: Keyword Mutation Candidates
 1: for all fields f of class C do
 2:     if f's access modifier contains volatile then
 3:         add f to the list volatileFields
 4:     end if
 5: end for
 6:
 7: for all methods m of class C do
 8:     if m's access modifier contains static then
 9:         add m to the list staticMethods
10:     else
11:         add m to the list nonStaticMethods
12:     end if
13:
14:     if m's access modifier contains synchronized then
15:         add m to the list synchronizedMethods
16:     else
17:         add m to the list nonSynchronizedMethods
18:     end if
19: end for
```

Figure 5.2: Keyword Mutation Candidates Algorithm.

To detect the access modifiers of methods and fields, we inspect the first parameters of the methods *visitMethod* and *visitField*, respectively. These first parameters define the access modifier of the given class elements and are of type integer. To extract the individual keywords from this parameter, we make use of the knowledge that all

access modifier keywords are defined to be positive powers of 2 in Java bytecode. The access modifier of a class element is sum of the integers its keywords correspond to. For instance, access value of the method in Figure 5.3 equals to 33 since *ACC_PUBLIC* is 1, and *ACC_SYNCHRONIZED* is 32. Another example can be seen in Figure 5.4 where the field x has the access value 66, because *ACC_PRIVATE* has the value 2, and *ACC_VOLATILE* has the value 64.

```
1 public class SUT {
2         public synchronized void myMethod(){
3                 // Method body
4         }
5 }
```

Figure 5.3: A method with access modifiers *public* and *synchronized*.

```
1 public class SUT {
2     private volatile int x;
3 }
```

Figure 5.4: A field with access modifiers *private* and *volatile*.

5.1.2. Critical Region Mutation Candidates

After all KW mutation candidates are identified, we go in deeper and start traversing methods' bytecodes. The algorithm to do this is given in Figure 5.5. Since we identify mutation candidates by the line number of their statement, we keep a variable *lineToMutate* and update it whenever we come across a line number instruction, as can be seen on lines 2-3.

On lines 4-8, we look for plain instructions whose opcodes are "MONITORENTER". These instructions define the beginning of a synchronized block.

5.1.3. Method Call Mutation Candidates

Concurrency in Java is obtained by not only access modifiers and synchronized blocks, but also some methods that provide communication between threads and reg-

```
Input: Java Method M
Output: Critical Region Mutation Candidates
 1: for all instruction insn of method M do
 2:     if insn is a line number instruction then
 3:         lastVisitedLine = insn.line
 4:     else if insn is a plain instruction then
 5:         if insn.Opcode == MONITORENTER then
 6:             add lastVisitedLine to syncBlocks list
 7:         end if
 8:     end if
 9: end for
```

Figure 5.5: Critical Region Mutation Candidates Algorithm.

ulate access to shared memory data. The algorithm to detect invocations of these methods can be seen on Figure 5.6. As well as inspecting the method call instructions, we are interested in line number instructions. This information is necessary to identify the line each method is called. Hence, we update the variable *lastVisitedLine* whenever we come across a line number instruction, on lines 2-3. On line 4, we check each instruction's type if it is a method call instruction. On lines 5-21, we check whether the method called is one of the *wait, join, sleep, yield, notify, notfiyAll, release* methods, and if it is, its invocation line is added to the method's corresponding list.

5.2. Mutant Creation

After all the mutation candidates are detected, Mutanticide creates actual mutants. Mutant creation process begins by critical region mutation. Then we create mutants related to the methods *sleep, wait, join, yield, notifyAll*, and *notify*. The mutant creation algorithm can be seen in Figure 5.7. Between lines 1-3 we create CR mutants based on the synchronized blocks data we collected. From line 4 to 24, we create MC mutants. Each type of method can be used in different types of mutants. For instance, a *join* method can be used to create RTXC, MXT, and RJS mutants as shown between lines 10-12, whereas only a RTXC mutant can be created from a *yield* method which can be seen between lines 13-15. Lastly, KW mutants are generated between lines 25 and 39.

Input: Java Method M
Output: Method Call Mutation Candidates

```
1: for all instruction insn of method M do
2:     if insn is a line number instruction then
3:         lastVisitedLine = insn.line
4:     else if insn is a method call instruction then
5:         if the method called is wait then
6:             add lastVisitedLine to the list methodWait
7:         else if the method called is join then
8:             add lastVisitedLine to the list methodJoin
9:         else if the method called is sleep then
10:            add lastVisitedLine to the list methodSleep
11:        else if the method called is yield then
12:            add lastVisitedLine to the list methodYield
13:        else if the method called is notify then
14:            add lastVisitedLine to the list methodNotify
15:        else if the method called is notifyAll then
16:            add lastVisitedLine to the list methodNotifyAll
17:        else if the method called is release and the method's owner is
18: java.util.concurrent.Semaphore then
19:            add lastVisitedLine to the list methodRelease
20:        end if
21:    end if
22: end for
```

Figure 5.6: Method Call Mutation Candidates Algorithm.

```
Input: Mutation Candidates
Output: Mutants
 1: for all elements of syncBlocks do
 2:     create EXCR, SKCR, SHCR, SPCR, RSB mutants
 3: end for
 4: for all elements of methodSleep do
 5:     create RTXC, MXT mutants
 6: end for
 7: for all elements of methodWait do
 8:     create RTXC, MXT mutants
 9: end for
10: for all elements of methodJoin do
11:     create RTXC, MXT, RJS mutants
12: end for
13: for all elements of methodYield do
14:     create RTXC mutants
15: end for
16: for all elements of methodNotifyAll do
17:     create RTXC, RNAN mutants
18: end for
19: for all elements of methodNotify do
20:     create RTXC, RNNA mutants
21: end for
22: for all elements of methodRelease do
23:     create RSR mutants
24: end for
25: for all elements of staticMethods do
26:     create RSTK mutants
27: end for
28: for all elements of nonStaticMethods do
29:     create ASTK mutants
30: end for
31: for all elements of synchronizedMethods do
32:     create RSK mutants
33: end for
34: for all elements of nonSynchronizedMethods do
35:     create ASTK mutants
36: end for
37: for all elements of volatileFields do
38:     create RVK mutants
39: end for
```

Figure 5.7: Create Mutants Algorithm.

5.2.1. ASK

To change a non-static method into static, we overload *ClassNode* class's method *visitMethod*. We add the constant *ACC_SYNCHRONIZED*, that equals to 32, to the first parameter that sets the method's access modifier.

5.2.2. ASTK

Since ASTK mutation is related with modification of a method declaration, we again overload the *visitMethod* method of *ClassNode* class. The first parameter to it is the access modifier of the method declared. Adding the constant *ACC_STATIC*, or 8, to this parameter makes the access modifier contain the static keyword.

5.2.3. RSK

RSK mutation is similar to ASK in terms of creation. Since we add the constant *ACC_SYNCHRONIZED* to the access modifier of the method declaration to create an ASK mutant, we subtract it for RSK.

5.2.4. RSTK

Just like RSK is the dual of ASK, RSTK is the dual of ASTK. Overloading the *visitMethod* method of *ClassNode* class, this time we subtract the constant *ACC_STATIC*, which corresponds to the value 8, from the original first parameter that defines the access modifier of the method declared.

5.2.5. RVK

Though it is related with fields instead of methods, RVK is another keyword mutation. In this case, we overload the method *visitField* of *ClassNode* class. This method, too, has access modifier as its first parameter. Subtracting the constant *ACC_VOLATILE*, of integer value 64, from the access modifier will leave us with a

non-volatile field if it was previously volatile.

5.2.6. EXCR

For EXCR mutation, we first check if the synchronized block in question has statements before and after it. If it has both, the block is expanded both sides. If it has statements only before or after itself, it is expanded upwards or downwards, respectively. If the method body has no other statements except for the synchronized block, no EXCR mutant is created.

5.2.7. SHCR

To create a SHCR mutant, we again check if it has any statements before and after. If there is a statement preceding the synchronized block, we shift it upwards. Or else, if it has a succeeding statement, we shift the block downwards. If there is neither a preceding nor a succeeding statement, we do not create a SHCR mutant of this synchronized block.

5.2.8. SKCR

Before creating a SKCR mutant, we count the number of lines in that synchronized block. If it consists of one line of statement, we skip the creation of this mutant. If it has two lines, we shrink it by pushing the first line out of the critical region. If it has three or more lines, we shrink from both sides and spit out the first and last statements of the synchronized block.

5.2.9. SPCR

For the case of SPCR mutation, we check if the critical region to mutate has two or more lines. If it does, we create the mutant, and leave it otherwise. In addition to changing the bytecode of the synchronized block, we add a try catch clock for the newly created critical region.

5.2.10. RSB

Creating a RSB mutant is quit straightforward compared to other critical region mutations. We do not need to check for previous and next lines of statements, or the number of lines inside the synchronized block. However, the try catch block associated with this synchronized block must also be removed from the method's instruction list.

5.2.11. MXT

To create an MXT mutant, when we are one the line to mutate, before adding the invocation instruction of the sleep method, we add a *LdcInsnNode* to load a constant with the value 2, then an *InsnNode* with the opcode *LDIV*. These additional instructions will cause the time parameter to be divided by 2. Alternatively, the opcode of the *InsnNode* can be assigned as *LMUL*, which will result in 2 times the original time parameter.

5.2.12. RJS

Before replacing a join method call with sleep, we declare a new method called "mySleep" which contains the instructions to call sleep method of the class Thread with a time parameter of 1000 milliseconds. Then, we substitute the join call with an invocation to this newly created method.

5.2.13. RTXC

An RTXC mutant can be produced by simply removing the instruction that is on the line the mutation candidate points at.

5.2.14. RNAN

The mutation RNAN is quite easy to produce. All we have to do is skip the method instruction node on the line to mutate, and add a new method instruction

instead. This new method call instruction will have “notify” as its name of the method to call.

5.2.15. RNNA

RNNA mutation production is almost same as the RNAN mutation. The only difference is, we add a method call instruction which has “notifyAll” as its name parameter. Thus, a call to notifyAll will be on the line where a call to notify method was.

5.2.16. RNNA

RSR mutation is removal of a method call which resembles RTXC. The method instruction node on the line pointed by the mutation candidate is skipped while writing the bytecode of the mutant.

6. EXPERIMENTS

We have run 4 sets of experiments on various benchmark programs. Some of these programs are large scale software such as junit [30] or components of apache-lang library [3]. The rest are experimental benchmark programs [20, 23] that come from multiple sources. Since our experiments are intensely rely on file operations, we used bash scripts. Additionally, making use of the control operator (&), bash scripts make it easier to run tests concurrently.

We apply *single mutation* in our experiments, which means we test one mutant at a time. This is necessary to distinguish the effects of each distinct mutant. The scripts also require the path to the directory in which the result files will be created, and any libraries to add to the classpath so that we can run the tests successfully.

The algorithm to run the experiments can be seen in Figure 6.1. Once we have all the file information, we clean the data by removing abstract and internal classes, on lines 3-9. Abstract classes are removed because they cannot be instantiated. Inner classes are deleted from the list since they cannot be tested on their own. On lines 11-13, we move the test classes to a separate list because they will not be mutated and will be used in a different manner.

When we have the class list in hand, we go into a loop on line 16, and pass each class to Mutanticide [38] one by one. On lines 17-18, our tool scans the class, detects mutation candidates and creates its mutants. Between lines 19-30, we run the entire test suite on each mutant. Meanwhile we create a text file in the result directory for the mutant. For each test run, name of the test is printed on a line, and the result of the test is printed as the following line. This procedure is repeated until all tests are complete. Then the next mutant in line is taken into the test loop and so on. After all mutants of a class are tested against the test suite, the mutants are removed from the system.

```
Input: Benchmark Program
Output: Test Results for all Mutants
 1: Scan classes of the benchmark up to a given bound of directory depth and add them to the array FILELIST
 2: for all classes C in the array FILELIST do
 3:     if C is an abstract class then
 4:         remove C from FILELIST
 5:     end if
 6:
 7:     if C is an inner class then
 8:         remove C from FILELIST
 9:     end if
10:
11:     if C is a test class then
12:         move C to TESTLIST
13:     end if
14: end for
15:
16: for all files in the array FILELIST do
17:     identify mutation candidates
18:     create mutants
19:     for all mutants created do
20:         for all tests in the array TESTLIST do
21:             run the test
22:             if test kills the mutant then
23:                 write "DEAD" to the mutant's result file
24:             else
25:                 write "ALIVE" to the mutant's result file
26:             end if
27:
28:             Wait for all tests to complete for this mutant
29:         end for
30:     end for
31:
32:     delete the mutants created for this class
33: end for
```

Figure 6.1: Run Mutation Testing Algorithm.

6.1. Experiment 1

In the first group of experiments we used medium scale applications in combination with experimental benchmark programs which can be seen in Table 6.1. The first column lists the names of the programs, the second column displays the number of lines of code each program has. The third and fourth columns list the number of Java classes and unit tests each program has, respectively.

The first two benchmark programs are components of *apache-lang* library [3]. Both the classes and the tests of *lang-concurrent* concentrate on concurrency mechanisms. On the other hand, *lang-build* specializes in creating classes with the builder design pattern. The last three benchmark programs in Table 6.1 are experimental software with small LoC values. *cmgtt* and *linkedlist* programs [20] are known to be buggy. They have the potential for stack overflow and data race, respectively. The program *fibonacci* [23] has no known bugs.

Table 6.1: Benchmark Programs for Experiment 1

Program	**LoC**	**#Classes**	**#Tests**
lang-concurrent	3105	14	10
lang-build	9758	16	23
cmgtt	496	7	1
linkedlist	420	4	1
fibonacci	63	1	1

Mutation testing results of the programs in Table 6.1 are shown in Table 6.2. Names of the programs are listed in the first column. The second column lists the number of mutants created for each program. The third column is the mutation score, which is the percentage of mutants killed by the test suite. The last column displays the total duration, which consists of finding mutation candidates and creating mutants from them, then running the test suite on each mutant.

Table 6.2: Results for Experiment 1

Program	**#Mutants**	**Mutation Score (%)**	**Total Duration** (minutes:seconds)
lang-concurrent	150	60.13	11:53
lang-build	386	40.51	191:47
cmgtt	56	44.06	01:07
linkedlist	40	38.63	01:09
fibonacci	14	78.57	00:13

When we compare the LoCs in Table 6.1 with the number of mutants in Table 6.2, we see that they are highly correlated. Number of classes and number of tests also strengthen this correlation. We can infer from the procedure of mutation testing that total duration of testing depends on the number of mutants and the number of tests in the test suite. Table 6.2 approves this inference.

The difference between the focuses of the first two benchmark programs results as the disparity of their mutation scores. *lang-concurrent* has a 1.5 times higher mutation score compared to *lang-build*. For the last three of the programs, we see that *fibonacci* has a higher mutation score than the other two. This can be elaborated by the fact that it has a lower LoC value and less classes, although all three have the same number of tests. Thus, *fibonacci*'s test suite becomes more sufficient compared to *cmgtt* and *linkedlist*.

Mutant count of the benchmark programs for each mutation operator is listed in Table 6.3. We see that ASK and ASTK mutants are dominant against other type of mutants, except for *fibonacci* which has a rather even distribution. From these numbers, we can infer that *fibonacci* makes use of concurrency mechanisms more than other programs do. This can be another reason why this program has the highest mutation score amongst these benchmark programs, since a concurrency bug affects the program output more severely and the test suite can kill those mutants more easily.

2-selective mutation results for the benchmark programs in this set are presented in Table 6.4. *linkedlist* is excluded in this set since it has no other mutants than of

Table 6.3: Mutant distribution for Experiment 1

	lang3-build	**lang3-concurrent**	**cmgtt**	**linkedlist**	**fibonacci**
ASK	201	58	27	20	4
ASTK	163	62	17	20	3
RSK	0	16	0	0	0
RSTK	22	12	10	0	1
RVK	0	1	0	0	0
EXCR	0	0	0	0	0
SHCR	0	0	0	0	0
SKCR	0	0	0	0	0
SPCR	0	0	0	0	0
RSB	0	0	1	0	0
MXT	0	0	0	0	0
RJS	0	0	0	0	3
RTXC	0	1	1	0	3
RNAN	0	0	0	0	0
RNNA	0	0	0	0	0
RSR	0	0	0	0	0
Total	386	150	56	40	14

Table 6.4: Selective Mutation for Experiment 1

Program	**#Mutants**	**Mutation Score (%)**	**Total Duration** (minutes:seconds)
lang-concurrent	30	46.66	01:49
lang-build	23	100.00	07:20
cmgtt	12	91.66	00:05
fibonacci	7	85.71	00:04

type ASK and ASTK. Selective mutation outcomes confirm with the inferences of [37] achieving a high mutation score with remarkable time improvements.

6.2. Experiment 2

The second experiment is run on a large scale benchmark program which has hundreds of classes, as seen in Table 6.5.

Results of this experiment is displayed in Table 6.6. Due to the size of the benchmark program, we applied partial mutation testing in this experiment. The number of mutants is very high, also. Majority of experiment duration was spent to test the mutants, all mutants were generated in slightly over 8 minutes.. JUnit also has a very comprehensive test suite that extends testing duration. However, when we inspect the structure of the software, we see that they are separated into different packages. Most large scale Java programs consist of a similar package tree. A reasonable and effective cost reduction technique would be running only the tests which are in the same package tree with the mutant class. Since a test in a different package has less chance of killing a mutant, a significant amount of time would be saved.

The time savings partial mutation brings is approximately 80%. A surviving mutant takes 23 seconds on average to run all the tests on it. Without partial mutation, this would be repeated for all mutants, whether dead or alive, which would equal to a testing duration of over 30 days.

Table 6.5: Benchmark Programs for Experiment 2

Program	**LoC**	**#Classes**	**#Tests**
JUnit	38618	195	200

Table 6.6: Results for Experiment 2

Program	**#Mutants**	**MS (%)**	**Mutant Generation** (minutes:seconds)	**Total Duration** (days:hours)
JUnit	115254	86.38	18:06	06:09

Mutant counts for each mutation type is listed in Table 6.7. We see that KW mutants make most of the mutant population. This is due to the fact that *JUnit* is a highly modular software, it consists of thousands of small methods. Nevertheless, we can say that it makes use of concurrency mechanisms by looking at the number of RSK, RVK, and RTXC mutants.

Another interesting point is the inequality of CR mutants. RSB has 12 mutants whereas other CR type mutations has 3 mutants. This is, again, due to the high modularity of this software. Most of the synchronized blocks consist of one line of statement and are the only code segment in their surrounding methods. Hence, those critical regions cannot be expanded, shifted, shrunk, or split.

Table 6.8: Selective Mutation for Experiment 2

Program	**#Mutants**	**MS (%)**	**Total Duration** (days:hours)
JUnit	16252	81.44	01:18

Table 6.7: Mutant distribution for Experiment 2

	JUnit
ASK	58389
ASTK	40613
RSK	1120
RSTK	13958
RVK	832
EXCR	3
SHCR	3
SKCR	3
SPCR	3
RSB	12
MXT	0
RJS	0
RTXC	318
RNAN	0
RNNA	0
RSR	0
Total	115254

2-selective mutation results of this experiment are shown in Table 6.8. Mutation score is acceptably close to the non-selective mutation score while testing duration is improved by 73%.

6.3. Experiment 3

Last set of experiments were run to assess the impact of our newly proposed mutation operator, RNNA. We know that this mutant could cause deadlocks or data

Table 6.9: Benchmark Programs for Experiment 3

Program	**LoC**	**#Classes**	**#Tests**
DiningPhilosophers	143	4	1
ProducerConsumer	188	4	1

Table 6.10: Results for Experiment 3

Program	**#Mutants**	**Mutation Score (%)**	**Total Duration** (minutes:seconds)
DiningPhilosophers	32	71.87	00:58
ProducerConsumer	46	67.39	02:23

races in the software. Due to the nature of concurrent programs, one cannot guarantee the same outcome each time they run it. Thus, if there is a probability of deadlock, it may or may not happen when they run the program once. Therefore, we ran mutation testing on this group of benchmark programs 10 times and took the union of dead mutants. Benchmark programs we used in this experiment can be seen in Table 6.9. *DiningPhilosophers* [19] and *ProducerConsumer* [45] are example problems of concurrent program synchronization.

In *DiningPhilosophers*, there are five imaginary philosophers who sit at a round table. There is fork between each pair of adjacent philosophers. Philosophers alternate between the states of *thinking*, *hungry*, and *eating*. A philosopher can eat when he holds both of the forks to his left and right, and each fork can be held by only one philosopher at a time. When a philosopher is *hungry*, he attempts to acquire 2 forks, and if he cannot, he waits till he can. He starts *eating* when he has both of the forks, and releases the forks later to go to the state of *thinking*.

The *ProducerConsumer* problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer is supposed to generate data and put it into the buffer until it is full, and start when it is available again. The consumer consumes the data by removing it from the buffer. The problem is to make sure that the producer does not produce data when the buffer is full and that the consumer does not try to consume data when the buffer is empty.

Table 6.11: Mutant distribution for Experiment 3

	DiningPhilosophers	**ProducerConsumer**
ASK	11	14
ASTK	10	14
RSK	2	4
RSTK	3	4
RVK	0	0
EXCR	0	0
SHCR	0	0
SKCR	0	0
SPCR	0	0
RSB	0	0
MXT	1	2
RJS	0	0
RTXC	4	6
RNAN	0	0
RNNA	1	2
RSR	0	0
Total	32	46

Table 6.12: 2-selective Mutation

Program	**#Mutants**	**Mutation Score (%)**	**Total Duration** (minutes:seconds)
DiningPhilosophers	11	42.85	00:47
ProducerConsumer	19	31.57	01:29

The result of this experiment can be seen in Table 6.10, and the number of mutants for each mutation operator is listed in Table 6.11. We see that ASK and ASTK mutation operators dominate the mutant pool. Thus, these mutants should be removed to apply 2-selective mutation.

The result of 2-selective mutation can be seen in Table 6.12. The percentage of time savings is less than the percentage of mutant count recession. This is due to the fact that ASK and ASTK mutation operators often produce *stillborn mutants*. These mutants are syntactically incorrect and fail to compile. Since we skip the compilation process by using bytecode instrumentation, these mutants are generated anyway and are instantly killed by the test suite because they fail to execute. In a real life software development process, these faults would not make it to the testing stage since they fail at the compilation phase.

In addition to n-selective mutation, we searched for a sufficient subset of mutation operators that kills a satisfactory percentage of all mutants while providing time improvements. Mutants created by a subset of operators and killed by the test suite make up a fraction F of all mutants. The aim of sufficient set problem is choosing a subset of operators so that F is close enough to the mutation score while providing time improvements. These sufficient subsets can be seen in Table 6.13. Our new operator RNNA appears in some of these sufficient subsets with considerable time improvements and acceptable mutation score cost.

Table 6.13: Sufficient set of mutation operators

	DiningPhilosophers		ProducerConsumer	
	Mutation Score (%)	Duration	Mutation Score (%)	Duration
All Operators	71.42	00:58	53.12	02:23
{ASK, ASTK, RSK, RSTK, RTXC}	66.67	00:41	50	01:40
{ASK, ASTK, RSK, RSTK, **RNNA**}	61.90	00:34	46.88	01:30
{ASK, ASTK, RSK, RTXC, **RNNA**}	61.90	00:40	46.88	01:35
{ASK, ASTK, RSTK, RTXC, **RNNA**}	61.90	00:33	43.75	01:26
{ASK, ASTK, RSK, RSTK}	57.14	00:32	43.75	01:23
{ASK, ASTK, RSK, RTXC}	57.14	00:38	43.75	01:28
{ASTK, RSK, RSTK, RTXC}	52.38	00:27	46.88	00:58
{ASK, ASTK, RSK, RSTK}	57.14	00:32	43.75	01:23
{ASTK, RSK, RSTK, **RNNA**}	47.62	00:20	43.75	00:48
{ASK, ASTK, RSK, RTXC}	57.14	00:38	43.75	01:28
{ASTK, RSK, RTXC, **RNNA**}	47.62	00:26	43.75	00:53
{ASTK, RSK, RSTK, RTXC}	52.38	00:27	46.88	00:58
{ASTK, RSK, RSTK, **RNNA**}	47.62	00:20	43.75	00:48
{ASTK, RSK, RTXC, **RNNA**}	47.62	00:26	43.75	00:53

Table 6.14: Benchmark Programs for Experiment 4

Program	**LoC**	**#Classes**	**#Tests**
SemaphoreTest	47	2	1

6.4. Experiment 4

We ran a small experiment on the experimental benchmark program shown in Appendix B. The stats of this benchmark program are shown in Table 6.14. The results are presented in Table 6.15. The mutation score was 46.15% without the RSR mutation operator and was improved to 57.14% with it. Mutant distribution is listed in Table 6.16.

Table 6.15: Results for Experiment 4

Program	**#Mutants**	**Mutation Score (%)**	**Total Duration** (minutes:seconds)
SemaphoreTest	14	57.14	00:29

Table 6.16: Mutant distribution for Experiment 4

	SemaphoreTest
ASK	4
ASTK	3
RSK	0
RSTK	0
RVK	0
EXCR	0
SHCR	0
SKCR	0
SPCR	0
RSB	0
MXT	0
RJS	0
RTXC	5
RNAN	0
RNNA	0
RSR	1
Total	14

7. CONCLUSION AND FUTURE WORK

In this work, we applied mutation testing to concurrent Java programs. We used our mutation operators RNNA and RSR with previously proposed 14 other operators. Since there was no publicly available tool for this purpose, we developed our own tool named "Mutanticide". We ran 4 sets of experiments with a different focus in each one.

In our first experiment, we observed that software which has tests specialized in concurrency has a higher mutation score.

In our second experiment, a large scale software achieved a high mutation score. However, high modularity of this software caused a dominance of some types of mutants over the rest. Another issue was the excessively long testing duration due to the high number of classes and tests. A workaround to this problem can be running only the tests that are in the same package tree with the mutant class. Since most large scale Java programs are divided into packages and the tests in a test suite focus on the classes in the same package, they have a much higher chance of killing those mutants than the tests in other packages.

In our third experiment, we inspected the effects of our newly proposed mutation operator, RNNA. We saw that this operator performed well for 2-selective mutation testing, as well as making it into some sufficient subsets of operators.

The mutation operator, RSR, we proposed focusing the programs that make use of the Java library *java.util.concurrent* was proven to be necessary in that group of programs and improved the mutation score when used in the operator pool.

7.1. Limitations

Our approach has several limitations. First of all, we cannot create CR mutants when there are more than one synchronized blocks in a method. Another similar

issue is, we cannot create CR mutants if a synchronized block is inside a control flow structure such as *if*, *while*, or *for*. This is due to the changing instruction labels in the bytecode when a jump instruction is displaced.

Another issue is the relatively small size of our benchmark pool. Although many software make use of concurrency, to demonstrate the effects of concurrent mutation operators better, we tried to select the ones with intensive concurrency mechanisms.

7.2. Future Work

Possible directions for future work include modifying Mutanticide to make use of mutant schemata generation. This way, the number of I/O operations would severely decrease and it would relieve the storage system.

Other than that, to improve the mutation testing duration, especially for large scale software, instead of running the whole test suite for each mutant, running only the tests that are in the same Java package with the mutant class would be a simple yet effective solution.

In addition, we are planning to extend the list of mutation operators for the Java library *java.util.concurrent* which has comprehensive concurrency mechanisms.

Lastly, we are planning to adapt mutation testing to Android systems using bytecode instrumentation. This requires proposing new mutation operators to reflect the faults that are specific to the Android programs and also commonly committed by Android programmers. Another issue on this matter is the type of virtual machine Android programs are run on. Android programs are compiled into Dalvik Executable (DEX) files which consist of instructions that are similar to the ones in a Java class file. However, DEX files are interpreted by the Android Runtime which is a register-based virtual machine, unlike JVM that is stack-based.

REFERENCES

1. J. H. Andrews, Y. Zhang. *General Test Result Checking with Log File Analysis.* IEEE Transactions on Software Engineering, vol 29, issue 7, 2003.

2. Apache Software Foundation. *Byte Code Engineering Library.* http://jakarta.apache.org/bcel, accessed at June 2016.

3. *Apache Commons Lang.* https://commons.apache.org/proper/commons-lang, accessed at June 2016.

4. *ASM.* http://asm.ow2.org/, accessed at June 2016.

5. A. Colyer, A. Clement, W. Isberg, M. Kersten, A. Vasseur, M. Webster, H. Hawkins, R Bodkin. *AspectJ.* https://eclipse.org/aspectj, accessed at June 2016.

6. E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi. *Toward the determination of sufficient mutant operators for C.* Software Testing, Verification and Reliability, vol. 11, issue 2, 2001.

7. *BCEL.* https://commons.apache.org/proper/commons-bcel/, accessed at June 2016.

8. J. S. Bradbury, J. R. Cordy, J. Dingel. *Mutation Operators for Concurrent Java (J2SE 5.0).* Second Workshop on Mutation Analysis (Mutation 2006), 2006.

9. J. S. Bradbury, J. R. Cordy, J. Dingel. *ExMan: A Generic and Customizable Framework for Experimental Mutation Analysis.* Second Workshop on Mutation Analysis (Mutation 2006), 2006.

10. E. Bruneton, R. Lenglet, T. Coupaye. *ASM: A code manipulation tool to implement adaptable systems*. Adaptable and Extensible Component Systems, 2002.

11. T. A. Budd, D. Angluin. *Two notions of correctness and their relation to testing*. Acta Informatica, vol 18, issue 1, 1982.

12. H. Coles. *PIT Mutation Testing Tool*. http://pitest.org/, accessed at June 2016.

13. J. Cordy, T. Dean, A. Malton, K. Schenider. *Source Transformation in Software Engineering Using the TXL Transformation System*. 1st IEEE International Workshop on Source Code Analysis and Manipulation, 2001.

14. M. Delahaye, L. Bousquet. *A Comparison of Mutation Analysis Tools for Java*. 13th International Conference on Quality Software, 2013.

15. R. Delamare, B. Baudry, Y. Le Traon. *AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors*. International Conference on Software Testing, Verification and Validation Workshops, 2009.

16. M. Delamaro, J. Maldonado. *Proteum - A Tool for the Assessment of Test Adequacy for C Programs*. Conference on Performability in Computing Systems, 1996.

17. R. A. DeMillo, R.J. Lipton, F.G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer*. Computer, Volume 11, Issue 4, 1978.

18. R. DeMillo, D. Guindi, K. King, W. McCracken, A. J. Offutt. *An Extended Overview of the Mothra Software Testing Environment*. Workshop on Software Testing, Verification, and Analysis, 1988.

19. *Dining philosophers problem*. https://en.wikipedia.org/wiki/Dining_philosophers_problem, accessed at June 2016.

20. Y. Eytani, K. Havelund, S. D. Stoller, S. Ur. *Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs*. Concurrency and Computation: Practice & Experience - Parallel and Distributed Systems: Testing and Debugging, vol 19, issue 3, 2007.

21. A. Flores, M. P. Usaola. *Testing-based assessment process for upgrading component systems* IEEE International Conference on Software Maintenance, 2008.

22. P. G. Frankl, S. N. Weiss, C. Hu. *All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness*. Journal of Systems and Software, vol 38, issue 3, 1997.

23. V. K. Garg. *Concurrent and Distributed Programming in Java*. John Wiley & Sons, 2004.

24. M. Gligoric, V. Jagannath, D. Marinov. *MutMut: Efficient Exploration for Mutation Testing of Multithreaded Code*. 3rd International Conference on Software Testing, Verification and Validation, 2010.

25. M. Gligoric, L. Zhang, C. Pereira, G. Pokam. *Selective Mutation Testing for Concurrent Code*. International Symposium on Software Testing and Analysis, 2013.

26. W. E. Howden. *Weak Mutation Testing and Completeness of Test Sets*. IEEE Transactions on Software Engineering, vol 8, no 4, July 1982.

27. S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, M. Utting. *Jumble Java Byte Code to Measure the Effectiveness of Unit Tests*. Testing: Academic and Industrial Conference Practice and Research Techniques, 2007.

28. *Java Path Finder*.
http://babelfish.arc.nasa.gov/trac/jpf, accessed at June 2016.

29. Y. Jia, M. Harman. *An Analysis and Survey of the Development of Mutation Test-*

ing. IEEE Transactions on Software Engineering vol 37, issue 5, 2011.

30. *Apache JUnit*.
http://junit.org/junit4/, accessed at June 2016.

31. R. Just. *The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java*. International Symposium on Software Testing and Analysis, 2014.

32. S. Kim, M. J. Harrold, Y. Kwon. *MUGAMMA: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution*. Second Workshop on Mutation Analysis (Mutation 2006), 2006.

33. K. N. King, A. J. Offutt. *A Fortran Language System for Mutation-Based Software Testing*. Software:Practice and Experience, vol. 21, no. 7, 1991.

34. Y. Ma, A. J. Offutt, Y. R. Kwon. *MuJava: An Automated Class Mutation System*. Journal of Software Testing, Verification and Reliability, vol 15, issue 2, June 2005.

35. L. Madeyski, N. Radyk. *Judy - A Mutation Testing Tool for Java*. IEEE Software, vol 4, issue 1, 2010.

36. P. R. Mateo, M. P. Usaola. *Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases*. 28th IEEE International Conference on Software Maintenance, 2012.

37. A. P. Mathur. *Performance, Effectiveness, and Reliability Issues in Software Testing*. 5th International Computer Software and Applications Conference, 1991.

38. *Mutanticide*.
https://github.com/koseus/mutanticide, accessed at June 2016.

39. A. S. Namin, J. H. Andrews, D. J. Murdoch. *Sufficient Mutation Operators for Measuring Test Effectiveness*.i 30th international conference on Software engineering, 2008.

40. A. J. Offutt, S. D. Lee. *An Empirical Evaluation of Weak Mutation*. IEEE Transactions on Software Engineering, vol 20, issue 5, May 1994.

41. A. J. Offutt, G. Rothermel, C. Zapf. *An experimental evaluation of selective mutation*. ICSE '93 Proceedings of the 15th international conference on Software Engineering, 1993.

42. A. J. Offutt, A. Lee. *An Experimental Determination of Sufficient Mutant Operators*. ACM Transactions on Software Engineering and Methodology, vol 5, issue 2, 1996.

43. A. Orso, D. Liang, M. Harrold, R. Lipton. *Gamma System: Continuous Evolution of Software after Deployment*. International Symposium on Software, Testing and Analysis, 2002.

44. M. Polo, M. Piattini, I. Garcia-Rodriguez. *Decreasing the Cost of Mutation Testing with Second-Order Mutants*. Software Testing, Verification and Reliability, vol 19, issue 2, 2008.

45. *Producer–consumer problem*.
https://en.wikipedia.org/wiki/Producer-consumer_problem, accessed at June 2016.

46. P. Reales, M. Polo, J. L. Fernandez-Aleman, A. Toval, M. Piattini. *Mutation Testing*. IEEE Software, vol 31, issue 3, May-June 2014.

47. A. Sen, M. S. Abadir. *Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing*. IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010.

48. *SERP*.
http://serp.sourceforge.net/, accessed at June 2016.

49. D. Schuler, A. Zeller. *Javalanche: Efficient Mutation Testing for Java*. The 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), August 2009.

50. R. H. Untch, A. J Offutt, M. J. Harrold. *Mutation Analysis Using Mutant Schemata*. International Symposium on Software Testing and Analysis, 1993.

51. M. P. Usaola, P. R. Mateo. *Mutation Testing Cost Reduction Techniques: A Survey*. IEEE Software, vol 27, issue 3, 2010.

52. P. J. Walsh. *A measure of Test Case Completeness*. PhD Dissertation, State University of New York at Binghamton, 1985.

53. J. A. Whittaker, J. Arbon, J. Carollo. *How Google Tests Software*. Addison-Wesley, 2012.

54. *Bytecode*.
https://en.wikipedia.org/wiki/Bytecode, accessed at June 2016.

55. L. Zhang, D. Marinov, L. Zhang, S. Khurshid. *Regression Mutation Testing*. International Symposium on Software Testing and Analysis, 2012.

56. L. Zhang, L. Zhang, S. Khurshid. *Injecting Mechanical Faults to Localize Developer Faults for Evolving Software*. ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications, 2013.

APPENDIX A: SAMPLE PROGRAM TO DEMONSTRATE RNNA

```
1  class ProducerConsumer {
2      static Consumer[] consumers = new Consumer[5];
3
4      public static void main(String[] args) {
5          BoundedBuffer buffer = new BoundedBuffer();
6          Producer producer = new Producer(buffer);
7
8          for(int i = 0; i < 5; i++){
9                          consumers[i] = new Consumer(buffer);
10         }
11         int satisfiedCount = 0;
12         while(satisfiedCount < 5){
13
14                 for (int i = 0; i < 5; i++)
15                         if(consumers[i].isSatisfied)
16                                 ++satisfiedCount;
17                 }
18         System.out.print(Consumer.total);
19         }
20     }
21 }
```

Figure A.1: Producer Consumer.

```
1  class Producer implements Runnable {
2  public static int totalProduced = 0;
3  BoundedBuffer b = null;
4
5  public Producer(BoundedBuffer initb) {
6  b = initb;
7  new Thread(this).start();
8  }
9
10 public void run() {
11 int item;
12 while (true) {
13 item = 1;
14
15 try {
16 b.deposit(item);
17 ++totalProduced;
18 } catch (InterruptedException e1) {}
19 try {
20 Thread.sleep(200);
21 } catch (InterruptedException e) {}
22 }
23 }
24 }
```

Figure A.2: Producer.

```
1  class Consumer implements Runnable {
2  public static int total = 0;
3  public int consumeCount = 0;
4  public boolean isSatisfied = false;
5
6  BoundedBuffer b = null;
7  public Consumer(BoundedBuffer initb) {
8  b = initb;
9  new Thread(this).start();
10 }
11
12 public void run() {
13 int item;
14 while (true) {
15 if(consumeCount == 5){
16 isSatisfied = true;
17 break;
18 }
19
20 try {
21 item = b.fetch();
22 ++consumeCount;
23 total += item;
24 } catch (InterruptedException e1) {}
25 try {
26 Thread.sleep(50);
27 } catch (InterruptedException e) {}
28 }
29 }
30 }
```

Figure A.3: Consumer.

```
1  class BoundedBuffer {
2      final int size = 10;
3      int[] buffer = new int[size];
4      int inBuf = 0, outBuf = 0;
5      BinarySemaphore mutex = new BinarySemaphore(true);
6      CountingSemaphore isEmpty = new CountingSemaphore(0);
7      CountingSemaphore isFull = new CountingSemaphore(size);
8
9      public void deposit(int v) throws InterruptedException{
10         isFull.P();// wait if buffer is full
11         mutex.P(); // ensures mutual exclusion
12         buffer[inBuf] = v; // update the buffer
13         inBuf = (inBuf + 1) % size;
14         mutex.V();
15         isEmpty.V();  // notify any waiting consumer
16     }
17     public int fetch() throws InterruptedException {
18         int value;
19         isEmpty.P(); // wait if buffer is empty
20         mutex.P();  // ensures mutual exclusion
21         value = buffer[outBuf]; //read from buffer
22         outBuf = (outBuf + 1) % size;
23         mutex.V();
24         isFull.V(); // notify any waiting producer
25         return value;
26     }
27 }
```

Figure A.4: Bounded Buffer.

```
1  public class CountingSemaphore {
2      int value;
3      public CountingSemaphore(int initValue) {
4          value = initValue;
5      }
6      public synchronized void P() throws InterruptedException{
7          value--;
8  //          System.out.println("buffer capacity: " + value);
9          if (value < 0){
10                         wait();
11         }
12     }
13     public synchronized void V() {
14         value++;
15 //          System.out.println("buffer capacity: " +  value);
16         if (value <= 0){
17                         notify();
18         }
19     }
20 }
```

Figure A.5: Counting Semaphore.

```
1  public class BinarySemaphore {
2      boolean value;
3      BinarySemaphore(boolean initValue) {
4          value = initValue;
5      }
6
7      public synchronized void P() throws InterruptedException{
8          while (value == false){
9              wait();// in queue of blocked processes
10         }
11         value = false;
12     }
13
14     public synchronized void V() {
15         value = true;
16         notify();
17     }
18 }
```

Figure A.6: Binary Semaphore.

APPENDIX B: SAMPLE PROGRAM TO DEMONSTRATE RSR

```
1  import java.util.concurrent.*;
2
3  public class SemaphoreTest {
4
5      public static void main(String args[]) {
6          Semaphore semaphore=new Semaphore(2);
7          MyThread mt1 = new MyThread(semaphore);
8          MyThread mt2 = new MyThread(semaphore);
9          MyThread mt3 = new MyThread(semaphore);
10         MyThread mt4 = new MyThread(semaphore);
11         mt1.start();
12         mt2.start();
13         mt3.start();
14         mt4.start();
15     }
16 }
```

Figure B.1: Driver Program.

```
1  import java.util.concurrent.Semaphore;
2
3  class MyThread extends Thread {
4
5      Semaphore semaphore;
6
7      MyThread(Semaphore semaphore){
8          this.semaphore = semaphore;
9      }
10
11     public void run() {
12         try {
13             semaphore.acquire();
14             System.out.println("Acquired " + this.getName());
15             try {
16                 sleep(2000);
17             } catch (Exception e) {}
18         } catch (InterruptedException ie) {
19         } finally {
20             semaphore.release();
21             System.out.println("Released " + this.getName());
22         }
23     }
24 }
```

Figure B.2: Sample Thread.