# A Test Suite Minimization Technique for Testing Numerical Programs

Prashanta Saha
*Gianforte School of Computing*
*Montana State University*
Bozeman, Montana, USA
prashantasaha@montana.edu

Clemente Izurieta
*Gianforte School of Computing*
*Montana State University*
*Idaho National Laboratories*
Bozeman, Montana, USA
clemente.izurieta@montana.edu

Upulee Kanewala
*School of Computing*
*University of North Florida*
Jacksonville, Florida, USA
upulee.kanewala@unf.edu

***Abstract*—Metamorphic testing is a technique that uses metamorphic relations (i.e., necessary properties of the software under test), to construct new test cases (i.e., follow-up test cases), from existing test cases (i.e., source test cases). Metamorphic testing allows for the verification of testing results without the need of *test oracles* (a mechanism to detect the correctness of the outcomes of a program), and it has been widely used in many application domains to detect real-world faults. Numerous investigations have been conducted to further improve the effectiveness of metamorphic testing. Recent studies have emerged suggesting a new research direction on the generation and selection of source test cases that are effective in fault detection. Herein, we present two important findings: i) a mutant reduction strategy that is applied to increase the testing efficiency of source test cases, and ii) a test suite minimization technique to help reduce the testing costs without trading off fault-finding effectiveness. To validate our results, an empirical study was conducted to demonstrate the increase in efficiency and fault-finding effectiveness of source test cases. The results from the experiment provide evidence to support our claims.**

***Index Terms*—Metamorphic testing, metamorphic relation, test case generation, test suite minimization, mutation testing**

## I. Introduction

Metamorphic Testing (MT) is a technique used to alleviate the *oracle* problem of software under test (SUT) [1]. A *test oracle* is a mechanism used to detect the correctness of the outcomes of a program [2]. In most cases of software behavior, it is easier to predict relationships between the elements of the output of a program, than to characterize the precise output given some input. For example, consider a program that computes the average of a list of real numbers. It is hard to correctly predict the observed output when the input has infinite possibilities. Thus, using this approach, we cannot validate whether the returned average is correct. However, we can permute the list of real numbers used in the input, and check to see if the returned output matches the output from the original test. If the outputs do not match, then there is a potential 'bug' (fault) in the program. This type of property is called a *metamorphic relation* (MR), a necessary property of the SUT that specifies a relationship between multiple test inputs and their outputs [3]. Thus, from existing test cases (i.e., source test cases) MRs are used to generate new test cases (i.e., follow-up test cases). The set of source and follow-up test cases are then executed on the SUT and the outputs are checked according to the corresponding MRs. The SUT can be considered faulty if an MR is violated.

MT has been successful in finding bugs in systems across various domains and has been successfully applied to detecting previously unknown faults in different domains such as web services, computer graphics, simulation and modeling, embedded systems etc. [4]. To date, work done on improving the fault detection effectiveness of MT has mainly focused on developing quality MRs [4]. However, developing such MRs is a labor-intensive task that requires the involvement of domain experts. Another, avenue to improve the fault detection effectiveness of MT, which has not, to our knowledge, been explored thus far, is to systematically generate the source test cases. In fact, most of the previous studies in MT have used randomly generated test cases or existing test cases as source test cases when conducting MT [5]–[8], [12]. Our previous work showed that the effectiveness of MT can be improved by systematically generating the source test cases based on some coverage criteria such as line, branch, and weak mutation (WM) [9]. But, it is sub-optimal to use a combination of all the coverage-based techniques to test numerical programs. A problem arises because test cases generated based on separate coverage criteria can be redundant, which means that there can be test case overlaps with the same code coverage as well as the same mutant killing rate making this approach inefficient.

In our research, we selected numerical programs as our target SUT since this domain has been relatively unexplored in the MT research [4]. The goal of this research is to select efficient and effective coverage-based test suites. To achieve this goal, we have divided our approach into two parts. First, we perform mutation analysis using a reduced set of mutants. Mutation analysis is necessary since we will apply it as our performance measurement metric to measure the fault-finding effectiveness of our test suites. This approach reduces the time and budget of the entire testing process. Second, we select the best coverage-based test suites among line, branch, and WM based on their cost-effectiveness and fault-finding effectiveness. In this way, we find the set of test suites with better efficiency and similar fault-finding effectiveness.

## II. Background

MT is a testing technique that aims to alleviate the oracle problem. However, the effectiveness of MT not only depends on the quality of MRs but also on source test cases. In this section, we discuss MT and source test case generation techniques. Specifically, we discuss line, branch, and WM coverage.

### A. Metamorphic Testing

Source test cases are used in MT to generate follow-up test cases using a set of MRs identified for the program under test (PUT) [10]. MRs are identified based on the properties of the problem domain, such that said properties uniquely identify some expected behavior of the problem [11]. For example, there exist some unique characteristics of weather systems that help testers to find the correct MRs. We can create source test cases using techniques like random testing [12], structural testing [13], or search-based testing [9]. Follow-up test cases are generated by applying the input transformation specified by the MRs. After executing the source and follow-up test cases on the PUT we can check to see if the MR was violated. The violation of the MR during testing indicates fault in the PUT. Since MT checks that the relationships between the inputs and outputs of a test program are maintained under the conditions of an MR, we can use this technique when the expected result of a test program is unknown. For example, in figure 1, a Java method *add_values* is used to illustrate how source and follow-up test cases work within a PUT. The *add_values* method aggregates the array elements passed as an argument. The source test case, $t = \{3, 43, 1, 54\}$ is randomly generated and tested on *add_values*. The output of this test case is 101. In this program, we would expect that when a constant $c$ is added to every element in the input collection, the output should increase accordingly. This expected behavior is used to generate an MR to conduct MT on this PUT. A constant value 2 is added to each element in the collection to create a follow-up test case $t^{'} = \{5, 45, 3, 56\}$ that is then run on the PUT. The output of the follow-up test case is 109. To satisfy this *Addition* MR, the follow-up test output should be greater than the source output. In this MT example, the *Addition* MR is satisfied for the given source and follow-up test cases.

### B. Coverage-based Test Case Generation

In this work, we used EvoSuite as the automated test case generation tool [14]. EvoSuite automatically generates test cases with coverage criteria e.g. line, branch, and WM approaches. EvoSuite uses an evolutionary search approach that simultaneously evolves test suites with respect to an entire coverage criterion. Below, we briefly describe the systematic approaches used by EvoSuite to generate coverage-based test suites.

***Line Coverage*** In Line coverage, to cover each statement of source code, it is required that each basic code block in a method is reached (except comments) [15]. In traditional search-based testing techniques, this reachability would be expressed by an association of branch distance and approach-level [17]. The branch distance measures how different a predicate (i.e., a decision-making point) is from evaluation to an expected target result. For example, given a predicate, *a == 7* and an execution where the value of *a = 5*, the branch distance to the predicate evaluating to true would be $|5 - 7| = 2$, whereas execution where the value of *a = 6* is closer to being true with a branch distance of $|6 - 7| = 1$. Branch distance can be estimated by applying a set of standard rules [17], [18]. The approach level measures the closest point of a given execution to the target node. If any test suite executes all the statements of a method, then the approach level will be 0, which means it will become insignificant.

***Branch Coverage*** Many popular tools have implemented in practice the idea of branch coverage, even though this practical approach may not always match the more theoretical interpretation of covering all edges of a program's control flow [14]–[16]. Branch coverage is often measured by maximizing the number of branches of conditional statements that are executed by a test suite. Thus, to satisfy a unit test suite for each of the branch statements, there is at least one test case that satisfies the branch predicate to *false* and at least one test case that satisfies the branch predicate to *true*. In branch coverage, the fitness function of a test suite is to cover all the branches in a method. This value is measured by calculating the closeness with which a test suite covers all the branches of a PUT.

***WM*** When generating test cases from test generation tools, the preferred practice is to satisfy the constraints or conditions (i.e. in WM when a test case reaches the mutated statement of a method) rather than a developer's preferred boundary cases [14]. In WM testing, small code modifications are applied to the PUT. Then, the test generation tools are forced to generate values that can distinguish between the original test case and the mutant test case. In mutation testing, a test case is considered "killed" when the execution result of the mutant version is different than the original version of the PUT. The WM criteria are satisfied when at least one test case from the unit test suite reaches the infection state of the mutant. To measure the fitness value of the WM, it is required to calculate the infection distance with respect to a set of mutation operators [15].

## III. Mutation Testing

Mutation testing has been used to evaluate the fault detection effectiveness of the automated test case generation approaches [19]. Mutation testing is a fault-based testing technique that measures the effectiveness of the test cases of the SUT. Many experiments suggest the usage of mutants as a proxy to real faults when comparing testing techniques [20]. Briefly, the testing technique follows these steps: First, mutants are created by simply seeding faults into a program. By applying syntactic changes to the original source code, new faulty versions of the original programs are generated. Each syntactic change is determined by an operator called a *mutation operator*. Test cases are then executed for the faulty and the original versions of the program and checked to

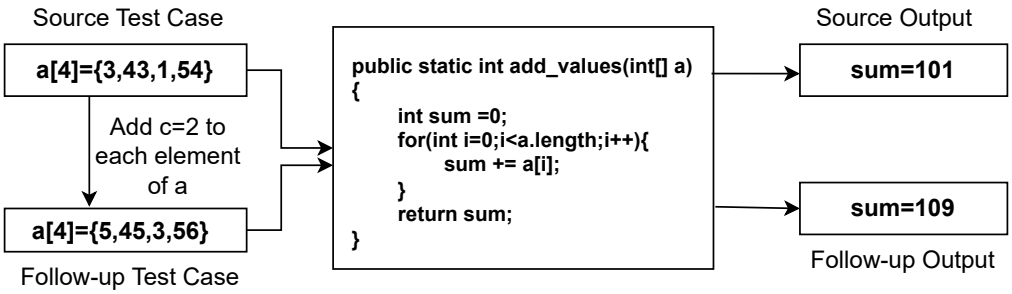


Fig. 1. Test Source and follow-up inputs on PUT.

see whether they produce different responses. If the response of the mutant is different from the original program, then we say that the mutant has been killed, and the test case is deemed to have the ability to detect faults effectively for that program. Otherwise, the mutant remains alive. When a mutant is syntactically different but semantically identical to the original program, it is referred to as an *equivalent mutant*. There are four common equivalent mutant situations: the mutant cannot be triggered, the mutant is generated from dead code, the mutant only alters the internal states of a program, and the mutant only improves the speed of execution of a program. The percentage of killed mutants to the total number of non-equivalent mutants provides an adequacy measurement of the test suite, which is called the *mutation score*.

## IV. Test Suite Minimization Approach

As systems evolve, their test suites are modified to accommodate new functionality. It is possible that redundant test cases (i.e., test cases for components that are already covered by other existing test cases) are introduced as test suites grow. Test suite minimization techniques address this problem by seeking to permanently remove redundant test cases in a test suite. The goal is to create a more efficient test suite, that is, smaller in size but effective at finding faults. The minimization process is typically accomplished without the knowledge of the changes in the new version of the program [21].

Our goal for this research is to select efficient and effective coverage-based test suites. To achieve this goal, we divided our approach into two parts. In the first part, we performed mutation analysis using a reduced set of mutants. Mutation analysis is necessary because we apply it as our performance measurement metric to assess the fault-finding effectiveness of our test suites. This approach saves the time and budget of the entire testing process. In the second part, we selected the best test suites among line, branch, and WM coverage based on their cost and fault-finding effectiveness. This allowed us to find reduced test suites with better efficiency and similar effectiveness.

### A. Mutants Reduction Approach

Selecting representative subsets from a given set of mutants is the principal aim of mutant reduction strategies. This technique reduces the application cost of mutation testing which leads to reductions in the total cost of software testing. Recent studies have shown two mutation reduction techniques that have proven to be highly effective [22]. But to the best of our knowledge, there are no mutation reduction techniques that have been applied to evaluate the fault-finding effectiveness of source test case generation techniques in MT. We claim that applying mutation reduction techniques to find better test case generation approaches is cost-effective in terms of time and budget. Thus, our goal is to find a better mutation reduction technique for our test case generation approach.

*1) Random Sampling Technique:* A major portion of the mutation testing demands is influenced by the generation and execution of the candidate set of mutants. By considering a small sample of mutants, a significant cost reduction can be achieved. Empirical studies have shown that a selection of 10% of mutants results in a 16% loss in the fault detection ability of the produced test sets when compared to full mutation testing [23]. In this study, we followed first-order mutation testing strategies [24] and selected a random x% portion of the initial mutants set, where $x = 10, 20, 30, 40, 50,$ and $60$. Our target was to find out which random % of selected mutants is a better representative of the total mutants set.

*2) Operator Based Mutant Selection:* Since mutant operators generate different numbers of mutant programs, Offutt et al. proposed $N - selective\ mutation$ theory, where $N$ is # of mutant operators [25]. In their experiment, they divided the mutant operators into three general categories based on the syntactic elements that they modify. The three categories are *Replacement-of-operand* operators (i.e., replace each operand in a program with each other legal operand), *Expression modification* operators (i.e., modify expressions by replacing operators and inserting new operators), and *Statement modification* operators (i.e., modify entire statements). Their experiments suggest that *Expression modification* operators with a smaller number of mutants than the total mutant set can be effective and the execution time is also shown to be linear. In this study, we applied *Expression modification* operators as mutants set to do the mutation analysis.

### B. Effective Test Suites Selection

In preliminary work, we showed that coverage-based test cases have better fault detection effectiveness than randomly generated test cases [26]. However, it is not feasible to use all

of the coverage-based techniques together to test numerical programs. This is because the process is time-consuming and test cases are repetitive regarding code coverage. Further, the fault detection effectiveness of test suites can vary based on the methods used. Therefore, we need an effective approach to help select better test suites when approaching SUT.

Mutation testing has been proven to be an effective approach to assessing the fault detection effectiveness of test case generation techniques. In our approach, we run mutation testing on the SUT and applied the test cases generated by the coverage-based test case generation techniques. After that, we measured the mutation score for each of the techniques. The test case generation technique with the highest mutation score was selected as a source test suite to test SUT.

## V. Empirical Evaluation

This section describes the design of the empirical evaluation approach: the research questions we will answer for our experiments, the description of the subject programs selected for the evaluation, description of the identified MRs for the subject programs and the evaluation process of the case study.

### A. Research Questions

- Research Question 1 (RQ1): Which Mutant reduction technique is best suited for detecting faults in MT?
- Research Question 2 (RQ2): Which coverage-based test suites have better fault-finding effectiveness?
- Research Question 3 (RQ3): Can test suite minimization techniques reduce the cost of executing a test suite and what is their effect on the fault detection effectiveness of a test suite?

### B. Subject Programs

We built a code corpus containing 96 methods that take numerical inputs and produce numerical outputs. We obtained these functions from the following open-source projects:

- **The Colt Project**[1]**:** A set of open source libraries written for high-performance scientific and technical computing in Java.
- **Apache Mahout**[2]**:** A machine learning library written in Java.
- **Apache Commons Mathematics Library**[3]**:** A library of lightweight and self-contained mathematics and statistics components written in Java.
- **Matrix.java:** This class has methods that perform matrix operations. We selected 20 methods randomly from the class and conducted our experiment on them. The description of these 20 methods is available in this GitHub repository[4].

Functions in the code corpus perform various calculations using sets of numbers such as calculating statistics (e.g., average, standard deviation, and kurtosis), calculating distances (e.g., Manhattan and Tanimoto), and searching/sorting. The total number of lines of code for these functions varied between 4 and 52, and the number of input parameters for each function varied between 1 and 4.

[1]http://acs.lbl.gov/software/colt/

[2]https://mahout.apache.org/

[3]http://commons.apache.org/proper/commons-math/

[4]https://github.com/ps073006/ConfRepo

### C. MR Identification

We selected all six MRs that were used in previous studies to test methods from the first 3 projects mentioned in section V-B [26]. Suppose our source test case is $X = \{x_1, x_2, x_3, ..., x_n\}$ where $x_i \geq 0, 0 \leq i \leq n$. Let source and follow-up outputs be $O(X)$ and $O(Y)$ respectively:

- **Addition:** add a positive constant C to the source test case yielding the follow-up test case $Y = \{x_1 + C, x_2 + C, x_3 + C, ..., x_n + C\}$. Then $O(Y) \geq O(X)$.
- **Multiplication:** multiply the source test case by a positive constant C yielding the follow-up test case $Y = \{x_1 * C, x_2 * C, x_3 * C, ..., x_n * C\}$. Then $O(Y) \geq O(X)$.
- **Shuffle:** randomly permute the elements in the source test case. The follow-up test case can then be $Y = \{x_3, x_1, x_n, ..., x_2\}$. Then $O(Y) = O(X)$.
- **Inclusive:** include a new element $x_{n+1} \geq 0$ in the source test case yielding the follow-up test case $Y = \{x_1, x_2, x_3, ..., x_n, x_{n+1}\}$. Then $O(Y) \geq O(X)$.
- **Exclusive:** exclude an existing element from the source test case yielding the follow-up test case $Y = \{x_1, x_2, x_3, ..., x_{n-1}\}$. Then $O(Y) \leq O(X)$.
- **Inversion:** take the inverse of each element of the source test case. Then the follow-up test case will be $Y = \{1/x_1, 1/x_2, 1/x_3, ..., 1/x_n\}$. Then $O(Y) \leq O(X)$.

We identified and developed the following ten MRs for testing the functions in the Matrix.java class. We have verified if the MRs satisfies each of the methods from the class and found out not all these MRs are satisfied by each of these methods. The entire list of methods and the specific MRs satisfied by them can be found in this GitHub repository[4]. In all cases, we assume that Matrix A comprises only non-negative numbers.

- **Scalar Addition:** Let $A$ be the initial input matrix to a program $P$, and $b$ be a positive scalar. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in b + A_{i,j}$. Let the output of $P$ for $A$ be $O$ (i.e. $P(A) = O$) and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.
- **Addition With Identity Matrix:** Let $A$ be the initial input matrix to a program $P$, and $I$ be an identity matrix. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in I_{i,j} + A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.
- **Scalar Multiplication:** Let $A$ be the initial input matrix to a program $P$, and $b$ be a positive scalar. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in b.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.
- **Multiplication With Identity Matrix:** Let $A$ be the initial input matrix to a program $P$, and $I$ be an identity

matrix. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in I_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **Transpose:** Let $A$ be the initial input matrix to a program $P$. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in A^T_{i,j} = A_{j,i}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.
- **Matrix Addition:** Let $A$ be the initial input matrix to a program $P$. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in A_{i,j}+A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.
- **Matrix Multiplication:** Let $A$ be the initial input matrix to a program $P$. Let $A'$ be the follow-up input matrix where $A' = \forall i, j \in A_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.
- **Column Permutation:** Let $A$ be the initial input matrix to a program $P$ with $j = 1, 2, 3, .., n$ columns. Let $A'$ be the follow-up input matrix after permuting the column positions of $A$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.
- **Row Permutation:** Let $A$ be the initial input matrix to a program $P$ with $i = 1, 2, 3, .., n$ rows. Let $A'$ be the follow-up input matrix after permuting the row positions of $A$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.
- **Element Permutation:** Let $A$ be the initial input matrix to a program $P$ with $j = 1, 2, 3, .., n$ columns and $i = 1, 2, 3, .., n$ rows. Rows and columns have to be same size. Let $A'$ be the follow-up input matrix after permuting $A_{i,n}$ element with $A_{n,j}$ element. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

### D. Evaluation Approach

In our evaluation, we applied the $\mu$Java[5], and PIT[6] tools to systematically generate mutants for our subject programs. For the 96 methods from the five open-source java projects, we generated a total of 8330 mutated versions using the mutation tools. Mutant distributions for the five classes are shown in Table I. We ran mutation testing on the five classes and exclude the mutants that cause compilation errors, runtime exceptions, and equivalent mutants. We also manually verified all the MRs for each subject programs.

TABLE I
INDIVIDUAL CLASSES FROM FIVE OPEN SOURCE PROJECTS. WE SHOW COUNTS OF METHODS, MRS AND MUTANTS

| Class name | # Methods | # Mutants | # MRs |
|---|---|---|---|
| MethodsCollection2.java | 28 | 1875 | 6 |
| MethodsFromMahout.java | 5 | 409 | 6 |
| MethodsFromApacheMath.java | 18 | 2248 | 6 |
| MethodsFromColt.java | 25 | 2914 | 6 |
| Matrix.java | 20 | 884 | 11 |

[5]https://github.com/jeffoffutt/muJava
[6]https://pitest.org/

TABLE II
SAVINGS OBTAINED BY OPERATOR BASED MUTANT REDUCTION APPROACH

| Class name | Total Mutants | Selected Mutants | Percentage Saved |
|---|---|---|---|
| MethodsCollection2.java | 1875 | 818 | 56.37 |
| MethodsFromMahout.java | 409 | 163 | 60.15 |
| MethodsFromApacheMath.java | 2248 | 723 | 67.84 |
| MethodsFromColt.java | 2914 | 994 | 65.89 |
| Total | 7446 | 2698 | 63.77 |

## VI. RESULTS AND DISCUSSIONS

Below we discuss the results of our experiments and provide answers to our research questions:

**RQ1. Which Mutant reduction technique is best suited for detecting faults in MT?** Figure 2 displays the average mutation scores (in %) per mutant reduction strategies for each class from SUT. The columns of Table II "Total Mutants" and "Selected Mutants" allow us to derive the mutant reduction percentage shown in the third column. We excluded Matrix.java class while analyzing the answer of RQ1 because we utilized the PIT tool to generate mutants for this class and this tool does not provide the mutant operators' names. So, we could not distinguish expression modification operators from the mutants set. The most interesting aspect of the figure 2 is that the operator-based mutation strategy always detects more faults than any other strategy and in the majority of cases this situation is statistically significant (p-value$<$0.05) (pairwise T-test results are available here[7]). Although testing with total mutants set (7446 mutants) has a comparatively high mutation score than randomly selected mutants set, but it is not as high as the operator-based mutation strategy. Mutation strategies where randomly selected mutants are generated (ranging from 10% to 60%) have comparatively low mutation scores.

Table II shows the savings obtained using the operator-based mutation strategy in terms of the number of mutants. The column "Percentage Saved" was computed by subtracting the number of "Selected Mutants" from the number of "Total Mutants" and dividing the difference by the number of "Total Mutants." The operator-based mutation strategy sets save anywhere between 56% to 67% of the total mutants that are generated across the subject programs. The expression modification operators (e.g., AORB, AORS, LOR, ROR, AOIU, COI, LOI) from the $\mu$Java tool are used for the operator-based mutation strategy.

**RQ2. Which coverage-based test suites have better fault-finding effectiveness?** Figure 3 shows the average mutation scores of the coverage-based test suites (Line, Branch, and WM) generated for the MT to test the five subject programs. To answer this RQ we combined the mutation scores of MT (source & follow-up test cases) for each subject programs.

[7]https://github.com/ps073006/ConfRepo

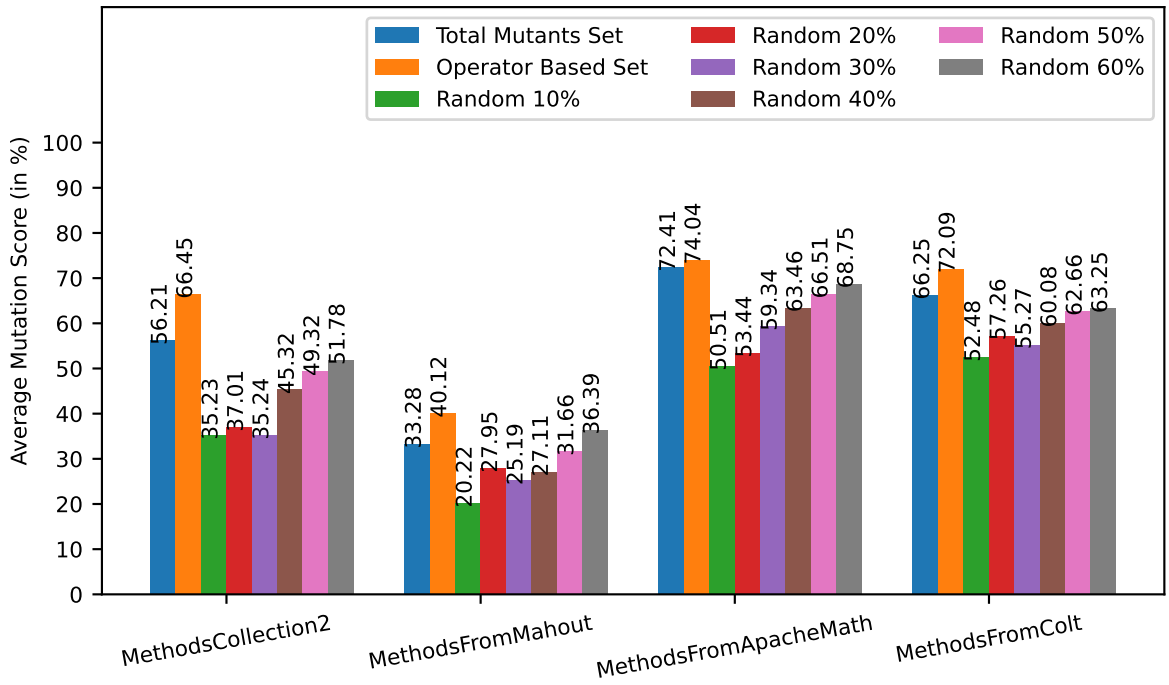


Fig. 2. Average Mutation Scores of Mutation Testing Strategies. Mutation Testing was done by the following strategies: Total Mutants Set, Operator based Set, Random 10%, Random 20%, Random 30%, Random 40%, Random 50%, and Random 60%. The average Mutation Score was calculated by averaging the mutation scores of all the methods from each of the classes.

The most interesting aspect of the figure is that WM-based test suites always detect more faults than any other test suites and in the majority of the subject programs. These results are statistically significant (p-value<0.05)(pairwise T-test results are available here[8]). However, in MethodsfromColt.java class, the difference in mutation scores is really small between the branch and WM test suites.

**RQ3. Can test suite minimization techniques reduce the cost of executing a test suite and what is their effect on the fault detection effectiveness of a test suite?** Table III reports the percentage reduction in test suite size for each of the five subject classes for the combined test suite of line, branch, and WM as well as WM coverage criteria separately. The results show that test suite minimization can significantly reduce the size of a test suite. The results also show that the WM coverage criteria yield a larger reduction in test suite size (53.13–65.43%). Results show that there is little variation in the reduction in test suite size, which means the performance of the WM test suites is linear across the subject programs.

Table IV reports the percentage reduction in fault detection effectiveness and code coverage of the WM coverage criteria. Minimized test suites of the WM coverage criteria perform well overall (1.44-14.14%) with fault detection effectiveness reduction as compared to the combined test suites. Also, the performance remains the same (0% reduction) for the code coverage with the minimized test suites of the WM coverage criteria as compared to the combined test suites.

Minimized test suites with WM can produce significant reductions in test suite size, resulting in significant savings in test execution costs. When using WM coverage criteria as a source test case generation technique for MT, test suite minimization causes an average reduction in fault detection effectiveness that is less than 9%. This makes the approach potentially useful in practice when testing time is limited and the system is not critical.

TABLE III
COST EFFECTIVENESS OF TEST SUITE MINIMIZATION TECHNIQUE FOR MT BASED ON TEST SUITE SIZE

| | Total Test Suite Size | | |
|---|---|---|---|
| Class Name | Combined | Weak Mutation | % Reduced |
| MethodsCollection2.java | 111 | 47 | 57.66 |
| MethodsFromMahout.java | 32 | 15 | 53.13 |
| MethodsFromApacheMath.java | 104 | 44 | 57.69 |
| MethodsFromColt.java | 81 | 28 | 65.43 |
| Matrix.java | 37 | 14 | 62.16 |

## VII. THREATS TO VALIDITY

We have followed Wohlin et al. guidelines while discussing the threats to the validity of our empirical study [27].

[8]https://github.com/ps073006/ConfRepo

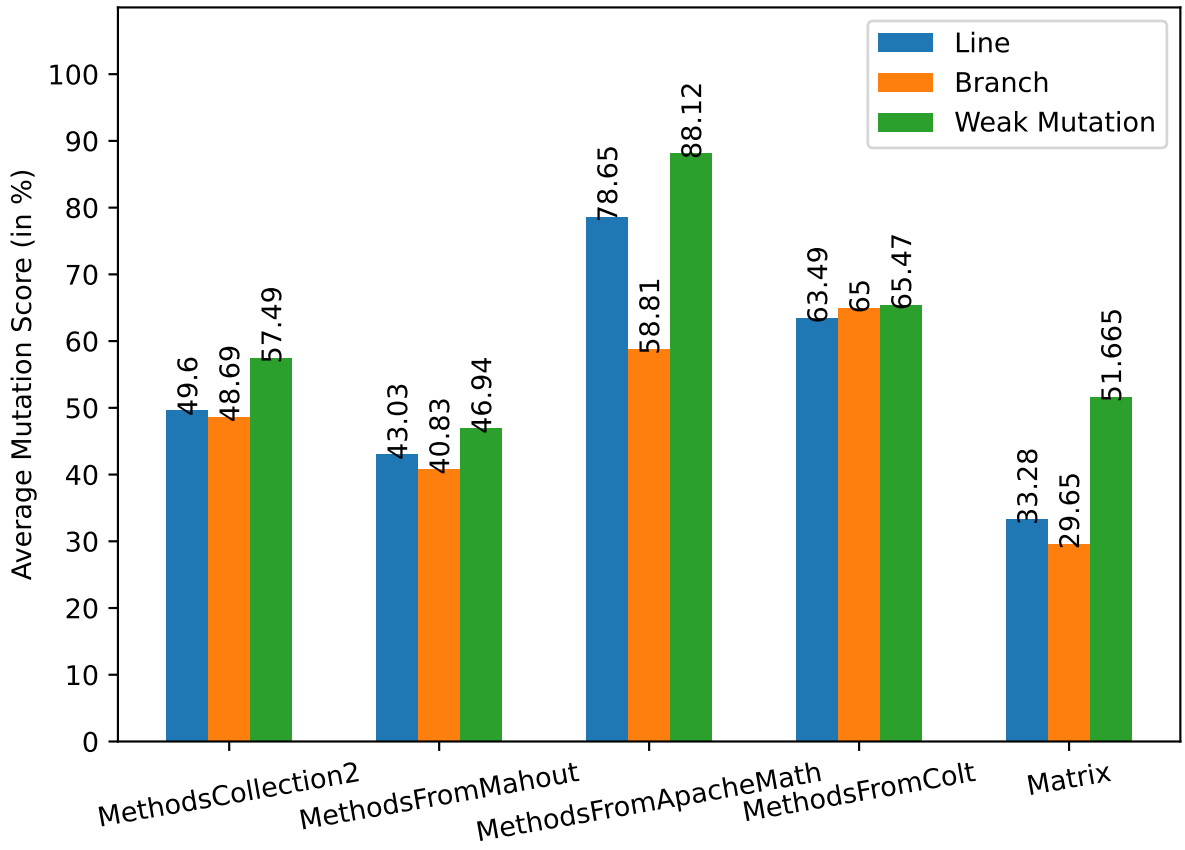


Fig. 3. Comparison of Average Mutation Scores of Coverage Based Test Suites, e.g. Line, Branch, and Weak Mutation. The average Mutation Score was calculated by averaging the mutation scores of all the methods from each of the classes.

TABLE IV
PERCENTAGE REDUCTION ON FAULT DETECTION AND CODE COVERAGE AFTER APPLYING TEST SUITE MINIMIZATION TECHNIQUE IN MT

| | Code Coverage | | | Fault Detection (Mutation score %) | | |
|---|---|---|---|---|---|---|
| Class Name | Combined | WM | % Reduced | Combined | WM | % Reduced |
| MethodsCollection2.java | 99.7 | 99.7 | 0 | 64.21 | 57.49 | 10.47 |
| MethodsFromMahout.java | 99.32 | 99.32 | 0 | 54.28 | 46.94 | 13.52 |
| MethodsFromApacheMath.java | 98.4 | 98.4 | 0 | 89.41 | 88.12 | 1.44 |
| MethodsFromColt.java | 99.16 | 99.16 | 0 | 76.25 | 65.47 | 14.14 |
| Matrix.java | 97.49 | 97.49 | 0 | 53.71 | 51.67 | 3.8 |

Threats to *internal validity* cause and effects may result from the way in which the empirical study was carried out. To increase our confidence in the experimental setup and mitigate this threat, we ran our experiments 10 times with the same setup.

Threats to *construct validity* may occur because of the third-party tools we have used. The EvoSuite tool was used to generate source test cases for line, branch, and WM test generation techniques. Further, we used the $\mu$Java and PIT mutation tool to create mutants for our experiment. To minimize these threats we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Threats to *external validity* were minimized by using the 96 methods from 5 different open-source project classes. This provides high confidence that the generalization of our results to other open-source software is appropriate. We only used the EvoSuite tool to generate test cases for our major experiment. But we also used the JCUTE [9] tool to generate branch coverage-based test suites for our initial case study and

[9]https://github.com/osl/jcute

also observed similar results [16].

## VIII. Conclusion

This paper presented an empirical study on the effects of test suite minimization on MT. Also, we present a hybrid approach to the reduced mutants set finding and the reduction in fault-finding effectiveness problems on numerical programs. The study used open source java projects which have been applied in previous studies, and test suites created using a systematic state-of-the-art approach. The mutated version of the code can be comparable with real-world faults.

The study showed that an operator-based mutant reduction technique can significantly reduce the mutant set size for mutation testing. This technique also keeps the mutation score comparable to the original mutant set. Practically this approach helps to reduce the total testing costs. However, we still need to perform industrial case studies to continue to scale the solutions presented in this work. Industrial case studies are still required to increase the power of these results.

The results also revealed that using minimized test suites with WM coverage criteria for MT provides a trade-off between the reduction in execution cost (53.13–65.43%) and the reduction in fault finding effectiveness (1.44–14.14%) that might be suitable in certain contexts in non-critical systems, where testing time and resources are limited.

Surprisingly, to date, there are no case studies that report on the impact of test suite minimization on fault detection effectiveness for MT. But, there were few case studies reported on test case prioritization for MT. This empirical study is the elemental step in this direction. Our future goal is to apply our proposed approach on real fault based programs such as Defects4J [10].

## References


[1] Chen, Tsong Yueh et al. “Metamorphic Testing: A New Approach for Generating Next Test Cases.” ArXiv abs/2002.12543 (2020): n. pag.

[2] Elaine J. Weyuker, On Testing Non-Testable Programs, The Computer Journal, Volume 25, Issue 4, November 1982, Pages 465–470, https://doi.org/10.1093/comjnl/25.4.465

[3] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. ACM Comput. Surv. 51, 1, Article 4 (January 2019), 27 pages. https://doi.org/10.1145/3143561

[4] S. Segura, G. Fraser, A. B. Sanchez and A. Ruiz-Cortés, ”A Survey on Metamorphic Testing,” in IEEE Transactions on Software Engineering, vol. 42, no. 9, pp. 805-824, 1 Sept. 2016, doi: 10.1109/TSE.2016.2532875.

[5] A. Gotlieb and B. Botella, ”Automated metamorphic testing,” Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003, Dallas, TX, USA, 2003, pp. 34-40, doi: 10.1109/CMPSAC.2003.1245319.

[6] Wu, Peng. “Metamorphic Testing and Special Case Testing: A Case Study.” Journal of Software 16 (2005): 1210.

[7] Sergio Segura, Robert M. Hierons, David Benavides, Antonio Ruiz-Cortés,Automated metamorphic testing on the analyses of feature models,Information and Software Technology,Volume 53, Issue 3, 2011,Pages 245-258.

[8] A. C. Barus, T. Y. Chen, F. -C. Kuo, H. Liu and H. W. Schmidt, ”The Impact of Source Test Case Selection on the Effectiveness of Metamorphic Testing,” 2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET), Austin, TX, USA, 2016, pp. 5-11, doi: 10.1145/2896971.2896977.

[9] P. Saha and U. Kanewala, ”Fault Detection Effectiveness of Source Test Case Generation Strategies for Metamorphic Testing,” 2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET), Gothenburg, Sweden, 2018, pp. 2-9.

[10] T. Y. Chen, ”Metamorphic Testing: A Simple Method for Alleviating the Test Oracle Problem,” 2015 IEEE/ACM 10th International Workshop on Automation of Software Test, Florence, Italy, 2015, pp. 53-54, doi: 10.1109/AST.2015.18.

[11] T. Y. Chen, F. -C. Kuo, D. Towey and Z. Q. Zhou, ”Metamorphic Testing: Applications and Integration with Other Methods: Tutorial Synopsis,” 2012 12th International Conference on Quality Software, Xi’an, China, 2012, pp. 285-288, doi: 10.1109/QSIC.2012.21.

[12] Chen, Tsong Yueh et al. “Metamorphic Testing and Testing with Special Values.” Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (2004).

[13] Junhua Ding, Tong Wu, Jun Q. Lu, and Xin-Hua Hu. 2010. Self-Checked Metamorphic Testing of an Image Processing Program. In Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI ’10). IEEE Computer Society, USA, 190–197. https://doi.org/10.1109/SSIRI.2010.25

[14] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software (ESEC/FSE ’11). ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[15] Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A. (2015). Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: Barros, M., Labiche, Y. (eds) Search-Based Software Engineering. SSBSE 2015. Lecture Notes in Computer Science(), vol 9275. Springer, Cham. https://doi.org/10.1007/978-3-319-22183-0_7

[16] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In Proceedings of the 18th international conference on Computer Aided Verification (CAV’06). Springer-Verlag, Berlin, Heidelberg, 419–423. https://doi.org/10.1007/11817963_38

[17] McMinn, P. (2004), Search-based software test data generation: a survey. Softw. Test. Verif. Reliab., 14: 105-156. https://doi.org/10.1002/stvr.294

[18] B. Korel, ”Automated software test data generation,” in IEEE Transactions on Software Engineering, vol. 16, no. 8, pp. 870-879, Aug. 1990, doi: 10.1109/32.57624.

[19] R. A. DeMillo, R. J. Lipton and F. G. Sayward, ”Hints on Test Data Selection: Help for the Practicing Programmer,” in Computer, vol. 11, no. 4, pp. 34-41, April 1978, doi: 10.1109/C-M.1978.218136.

[20] J. H. Andrews, L. C. Briand and Y. Labiche, ”Is mutation an appropriate tool for testing experiments? [software testing],” Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., St. Louis, MO, USA, 2005, pp. 402-411, doi: 10.1109/ICSE.2005.1553583.

[21] W. E. Wong, J. R. Horgan, A. P. Mathur and A. Pasquini, ”Test set size minimization and fault detection effectiveness: a case study in a space application,” Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC’97), Washington, DC, USA, 1997, pp. 522-528, doi: 10.1109/CMPSAC.1997.625062.

[22] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of Advances in Computers, pages 275–378. Elsevier, 2019.

[23] Weichen Eric Wong. 1993. On mutation and data flow. Ph.D. Dissertation. Purdue University, USA. Order Number: UMI Order No. GAX94-20921.

[24] M. Papadakis and N. Malevris, ”An Empirical Evaluation of the First and Second Order Mutation Testing Strategies,” 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, France, 2010, pp. 90-99, doi: 10.1109/ICSTW.2010.50.

[25] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.

[26] Prashanta Saha and Upulee Kanewala. 2018. Fault detection effectiveness of source test case generation strategies for metamorphic testing. In Proceedings of the 3rd International Workshop on Metamorphic Testing (MET ’18). Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/3193977.3193982

[27] C. Wohlin, M. Höst, P. Runeson, M. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering: an introduction. Kluwer Academic Publishers, 2000


[10] https://github.com/rjust/defects4j