applied sciences

Article

Empirical Comparison of Higher-Order Mutation Testing and Data-Flow Testing of C# with the Aid of Genetic Algorithm

Eman H. Abd-Elkawy [1,2] and Rabie Ahmed [1,2,*]

1 Department of Computer Science, Faculty of Computing & IT, Northern Border University, Arar 73213, Saudi Arabia; eman.hassan@nbu.edu.sa
2 Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Beni-Suef 62521, Egypt
* Correspondence: rabie.ahmed@nbu.edu.sa

Abstract: Data-Flow and Higher-Order Mutation are white-box testing techniques. To our knowledge, no work has been proposed to compare data flow and Higher-Order Mutation. This paper compares all def-uses Data-Flow and second-order mutation criteria. The comparison will support the testing decision-making, especially when choosing a suitable criterion. This compassion investigates the subsumption relation between these two criteria and evaluates the effectiveness of test data developed for each. To compare the two criteria, a set of test data satisfying each criterion is generated using genetic algorithms; the set is then used to explore whether one criterion subsumes the other criterion and assess the effectiveness of the test set that was developed for one methodology in terms of the other. The results showed that the mean mutation coverage ratio of the all du-pairs adequate test cover is 80.9%, and the mean data flow coverage ratio of the second-order mutant adequate test cover is 98.7%. Consequently, second-order mutation "ProbSubsumes" the all du-pairs data flow. The failure detection efficiency of the mutation (98%) is significantly better than the failure detection efficiency of data flow (86%). Consequently, second-order mutation testing is "ProbBetter" than all du-pairs data flow testing. In contrast, the size of the test suite of second-order mutation is more significant than the size of the test suite of all du-pairs.

Keywords: data-flow testing; higher-order mutation testing; "ProbSubsumes"; "ProbBetter"

Citation: Abd-Elkawy, E.H.; Ahmed, R. Empirical Comparison of Higher-Order Mutation Testing and Data-Flow Testing of C# with the Aid of Genetic Algorithm. *Appl. Sci.* **2023**, *13*, 9170. https://doi.org/10.3390/app13169170

Academic Editor: Christos Bouras

Received: 6 June 2023
Revised: 5 August 2023
Accepted: 9 August 2023
Published: 11 August 2023

1. Introduction

Data flow and mutation testing are two standard white-box testing methodologies [1]. The two methodologies are more effective and widespread than the other white testing criteria, such as statement and branch coverage [2]. Additionally, data flow and mutation coverage criteria subsume statement and branch coverage requirements [2].

There are only a few studies have been presented to compare and evaluate the differences between data flow and mutation testing concerning (1) effectiveness: number of faults detected by each approach; and (2) efficiency: number of test cases needed by each approach for creating an adequate test suite [2–5]. Some of the previous studies have analytically compared mutation and data flow [5,6], and other studies have empirically compared them [2–4,7].

To the best of our knowledge, the key study for comparing data flow and mutation testing techniques was proposed by Mathur and Wong in 1994 [3]. Mathur and Wong [3,8] manually designed test data to meet all-uses and mutation criteria and compare the scores to check the "ProbSubsumes" relationship between the two criteria. The experiment consisted of four programs and thirty groups of test data for each program. The results showed that mutation-cover was closer to satisfying data flow than data flow-cover was to mutation. In their study, Mathur and Wong compared two test adequacy criteria, called data flow-based all-uses criterion and mutation-based criterion. This research compared the hardness of fulfilling the two criteria and the cost of each of them. They showed that satisfying the mutation criterion was more challenging than all du-pairs.

Offutt and Voas [4] presented a comparison of mutation with all-defs data flow testing. They concluded that mutation cover subsumes all-defs cover (i.e., "any set of test cases that satisfies mutation will also satisfy all-defs"). The experiment consisted of ten programs and five groups of test data for each program generated using different methods. They also studied the fault detection ability of the generated tests.

Tewary and Harrold [9] developed techniques for seeding faults into the code under test utilizing the "program dependence graph." Then, they compared the fault-revealing capability of mutation and data flow. The results showed that the sufficient test sets for mutation and data flow were nearly equally successful in revealing the faults.

Offutt and Voas [4] concluded that no experiment could prove that one test approach is superior to another. Still, additional and dissimilar programs and studies can improve confidence in the validity of the relationship between the testing criteria.

Therefore, this paper introduces a new comparison between mutation and data flow testing. The significant contribution of this comparison is utilizing search-based algorithms to generate the test sets. In addition to the new test-data generation approach, this study uses a new set of C# programs different from the related work.

This paper is organized as follows. Section 2 presents some essential concepts about data flow and higher-order mutation testing for understanding this article. Section 3 introduces in detail the proposed empirical comparison. Section 4 gives the experiments and results. Finally, the conclusions and future work are given in Section 5.

2. Data Flow and Mutation Testing

Below are the basic concepts of data flow testing and higher-order mutation testing. Additionally, the expected relationships between the testing criteria are introduced.

2.1. Data Flow Testing

In any program, a limited number of actions can happen to a variable with the following events:

Definition: A statement loading a value in the memory space of the "variable" makes a definition (def) of that "variable".

Use: A statement reading a value from the memory slot of the "variable" is a use of the currently active definition of the "variable".

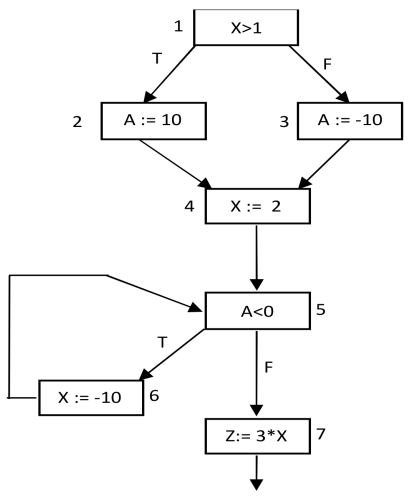
Data flow analysis [10] uses the control flow graph of the program to find the def-use pairs.

A program may be represented by a graph with "a set of nodes" and "a set of edges", called the control flow graph (CFG). Each "node" represents a statement, and each edge is a possible control flow between the nodes.

Defs and c-uses are associated with nodes, but p-uses are associated with edges. Def-use associations are represented by triples (v, s, u), where the value of variable v defined in statement s is used in statement or edge u.

In Figure 1, the triples (X, 4, 7) and (A, 2, (5,6)) represent def-use associations.

In order to determine the set of paths that satisfy the all-uses criterion, it is necessary to determine the defs of every variable in the program and the uses that might be affected by these defs [11].



Definition-c-use Associations

(X, 4, 7)

(X, 6, 7)

Definition-p-use Associations

(A, 2, (5, 6))

(A, 2, (5, 7))

(A, 3, (5, 6))

(A, 3, (5, 7))

Figure 1. Def-c-uses for the variable X and def-p-uses for the variable A.

2.2. *Higher-Order Mutation*

The mutation testing technique targets finding the test data, evaluating their efficiency, and simulating the other test coverage criteria [12,13]. The mechanism of mutation testing depends on injecting the targeted program with an error or more to obtain faulty versions called mutants [12]. A mutant is killed if the output of executing the program against a test datum differs from the output of the mutant. A mutant is alive if the output of executing the program and the mutant against all test data are the same in the current test suite. An alive mutant is "equivalent" if the output of executing the program and the mutant against all possible test data are the same.

The first-order mutation was suggested by DeMillo and Hamlet [14,15] via seeding a single error inside the program. Jia and Harman [12] proposed the Higher-Order Mutation via seeding double errors or more in the program. Table 1 introduces an example of the original code and first-, second-, and third-order mutants for this code.

Table 1. FOM and HOM examples.

Original Code	**Mutated Version**		
		Higher Order Mutant	
	First-Order	**Second-Order**	**Third-Order**
double n, m double d = n × m double d = n/m	double n, m double d = n + m double d = n/m	double n, m double d = n + m double d = n − m	double n, m double d = n + m double d = n − m++

The quality of data to detect errors is calculated using the mutation score (Equation (1)).

$$\text{Mutation Score} = \frac{\text{No. of killed mutants}}{(\text{ Total no. of mutants} - \text{No. of equivalent mutants})} \times 100 \quad (1)$$

2.3. *Relationship between Testing Criteria*

Weyuker, Weiss, and Hamlet [16] proposed a relationship between any two testing criteria called "ProbBetter". According to this relationship, "a testing criterion c1 is "ProbBetter" than c2 for a program P if a randomly selected test set T that satisfies c1 is more "likely" to detect a failure than a randomly selected test set that satisfies c2".

Mathur and Wong [3] proposed another relationship between any two testing criteria called "ProbSubsumes". According to this relationship, "a testing criterion c1 "ProbSubsumes" c2 for a program P if a test set T that is adequate with respect to c1 is "likely" to be adequate with respect to c2".

3. The Proposed Approach

In this section, we describe the phases that comprise the proposed approach. The system is written in C# and consists of the following modules:

1. Program analyzer.
2. Mutant generator.
3. Test data generator.

Figure 2 shows the architecture of the proposed approach. In the following, we will discuss these modules in detail.

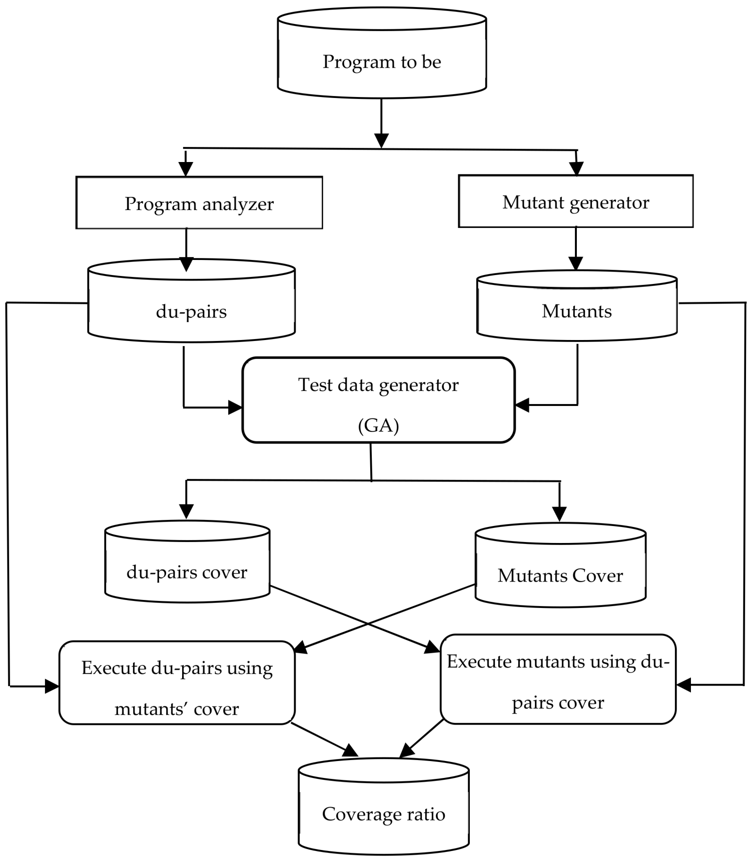


Figure 2. Overall chart of the proposed approach.

3.1. Program Analyzer Module

This module accepts the original program P in C# programming language as input. Then it analyses the original program to collect vital information to find the def-use associations. In addition, it instruments the original program with probes to identify the executed parts of the original program.

The output of this phase includes:

- An Instrumented version (P') of the given original program. The system instruments the assigned programs with software probes. During program execution, these probes cause the number of each traversed line to be recorded to keep track of the path traveled during the execution.
- The static analysis reports contain information about the components of each program: classes, objects, statements, variables, and functions.
- The control flow graph for the tested program.
- The list of variables def-use pairs for the tested program.

3.2. Mutant Generator Module

This module accepts the original program P in C# programming language as input. Then it passes the tested program into CREAM, which applies a set mutation operator to create a set of mutants. CREAM (Creator of Mutants) is a tool that introduces minor faults called mutations into C# programs. It is helpful to investigate the quality of a mutated program and the effectiveness of a prepared test suite. The CREAM system generates several object-oriented types of faults, puts them into the code of the original program according to its parsed trees, and compiles the modified sources. Obtained assemblies can be tested by measuring the number of revealed modifications. The output of this phase is a set of mutants.

3.3. Test Data Generator Module

Test-data generation in software testing identifies a set of program input data that satisfy a given test-coverage criterion. This phase uses the GA algorithm [17] to generate test cases to cover all def-use pairs of the given program or its mutant. Then, the tested program is executed with the test cases generated by the GA.

To apply the concepts of GAs to the problem of test-data creation, the GAs-based test-data generation techniques perform the following tasks:

- Consider the population to be sets of test data (test suite). Each set of test data is represented by a binary string called a chromosome.
- Find the sets of test data that represent the initial population. These sets can be randomly generated according to the format and type of data used by the program under test, or they may be input to the GAs.
- Determine the fitness of each individual in the population, which is based on a fitness function that is problem dependent.
- Select two individuals that will be combined in some way to contribute to the next generation.
- Apply the crossover and mutation processes.
- The input to this phase includes:
- Instrumented version of the program to be tested.
- List of def-use paths to be covered.
- Number of program input variables.
- Domain and precision of input data.
- Population size.
- Maximum no. of generations.
- Probabilities of crossover and mutation.

The output of this phase includes:

- A dynamic analysis report that shows the traversed path(s) and list of covered and uncovered def-use pairs after executing the given program with the test cases generated by the GA.
- Set of test cases (du pairs cover) that cover the def-use paths of the given program, if possible. The GA may fail to find test cases to cover some of the specified def-use paths when they are infeasible (i.e., no test data can be found by the GA to cover them).
- Set of test cases (mutants cover) that cover the mutant.

The algorithm evaluates each test case by executing the program with it as input and recording the def-use pairs in the program that are covered by this test case. The ratio between the number of covered def-use pairs by this test case to the total number of def-use pairs as a fitness function.

$$\text{fitness\_value}\,(\mathrm{v_i}) = \frac{\text{no. of def} - \text{use pairs covered by } \mathrm{v_i}}{\text{total no. of def} - \text{use pairs}} \tag{2}$$

GA uses a binary vector as a chromosome to represent values of the program input variables x. The length of the vector depends on the required precision and the domain length for each input variable.

Suppose we wish to generate test cases for a program of k input variables $x_1, \ldots, x_k$, and each variable x_i can take values from a domain $D_i = [a_i, b_i]$. Suppose further that d_i decimal places are desirable for the values of each variable x_i. To achieve such precision, each domain D_i should be cut into $(b_i - a_i) \cdot 10^{d_i}$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \cdot 10^{d_i} \leq 2^{m_i} - 1$. Then, a representation having each variable x_i coded as a binary string stringi of length mi clearly satisfies the precision requirement. The mapping from the binary string string_i into a real number x_i from the range $[a_i, b_i]$ is performed by the following formula:

$$x_i = a_i + x_i{'} \cdot \frac{b_i - a_i}{2^{m_i} - 1}, \tag{3}$$

where $x_i{'}$ represents the decimal value of the binary string stringi [17]. It should be noted that the above method can be applied for representing values of integer input variables by setting d_i to 0, and using the following formula instead of the formula in Equation (3):

$$x_i = a_i + \text{int}(x_i{'} \cdot \frac{b_i - a_i}{2^{m_i} - 1}), \tag{4}$$

Now, each chromosome (as a test case) is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$; the first m1 bits map into a value from the range $[a_1, b_1]$ of variable x_1, the next group of m_2 bits map into a value from the range $[a_2, b_2]$ of variable x_2, and so on; the last group of m_k bits map into a value from the range $[a_k, b_k]$ of variable x_k.

For example, let a program have two input variables, x and y, where $-3.0 \leq x \leq 12.1$ and $4.1 \leq y \leq 5.8$, and the required precision is four decimal places for each variable. The domain of variable x has a length of 15.1; the precision requirement implies that the range $[-3.0, 12.1]$ should be divided into at least $15.1 \cdot 10{,}000$ equal size ranges. This means that 18 bits are required as the first part of the chromosome: $2^{17} < 151{,}000 \leq 2^{18}$. The domain of variable y has a length of 1.7; the precision requirement implies that the range [4.1, 5.8] should be divided into at least $1.7 \cdot 10{,}000$ equal size ranges. This means that 15 bits are required as the second part of the chromosome: $2^{14} < 17{,}000 \leq 2^{15}$. The total length of a chromosome (test case) is then m = 18 + 15 = 33 bits; the first 18 bits code x and remaining 15 bits code y. Let us consider an example chromosome: 010001001011010000111110010100010.

By using the formula in Equation (3), the first 18 bits, 010001001011010000, represents x = 1.0524, and the next 15 bits, 111110010100010, represents y = 5.7553. So, the given chromosome corresponds to the data values 1.0524 and 5.7553 for the variables x and y, respectively.

4. The Experiments and Results

This section presents the experiment to assess the efficiency of the two testing criteria and their relationship. The investigation studies the "ProbBetter" and "ProbSubsumes" relationships of the two criteria: mutation and def-use. We run the proposed method (given in Figure 2) for the original codes of a selected set of C# programs.

4.1. Subject Programs

The subject programs were designated as a group of C# programs utilized in the experiments in numerous types of research [18,19]. Table 2 briefly describes the specifications of the subject programs.

Table 2. The subject programs.

Title	Description	Scale Using LOC (Lines of Codes)
Triangle	find the type of triangle according to its sides' lengths.	52 LOC
Sort	arrange a group of items	40 LOC
Stack	using push method to enter a group of elements and using pop method to delete element from stack.	51 LOC
Reversed	returns the array after inverting the elements	44 LOC
Clock	return the time in hours, minutes, and seconds	63 LOC
Quotient	return the quotient and the remainder of the division between two numbers	43 LOC
Product	find the summation, multiplication, and subtraction of three numbers.	54 LOC
Area	find the areas of any circle, triangle, or rectangle.	59 LOC
Dist	calling variables with more than one object and printing their dependent values with each call	52 LOC
Middle	find the middle value of three numbers.	42 LOC

4.2. Mutant Generator

The current work applied the CREAM [20] tool in the experiment to generate the mutants. Derezińska presented a set of specific C# mutation operators [21,22] and implemented those operators in a C# tool called CREAM [20].

4.3. GA Parameters Setup

The genetic algorithm parameters were set up after a set of trial runs. The parameters are experimental-based and determined by executing the genetic algorithm on a subset of the tested programs. GA settings are adapted to create a test suite of 20 tests (i.e., population size is 20), and GA iterates itself 100 times (i.e., the maximum number of generations is 100). It has been applied 30 times on the same machine for each subject program. The proposed GA used the single point crossover with probability = 0.8 and the flip bit mutation with probability = 0.15. After setting up the parameters of the genetic algorithms and the configurations of the CREAM tool, the following procedure was applied to each one of the subject programs given in Table 2.

Each subject program is passed to the analyzer to find the du-pairs and the mutant generator (the CREAM tool) to find the first-order and the second-order mutants. Table 3 gives:

1. The programs' titles (in column 1),
2. The number of first-order mutants (in column 2),
3. The number of killable first-order mutants (in column 3),
4. The number of second-order mutants (in column 4),
5. The number of killable second-order mutants (in column 5),
6. The number of killable second-order mutants that affect du-pairs (in column 6), and
7. The number of du-pairs for each subject program (in column 7).

This process is performed for each subject program as follows.

- Apply the CREAM tool to generate first order mutants.
- Generate the second-order mutants for each program by applying the CREAM tool on all first-order mutants.
- Eliminate the second-order mutants that are stillborn, equivalent, or do not contain a def or use for any du-pairs.
- Generate the def-uses pairs for each program.

- Eliminate the stillborn and the equivalent mutants from the first and second-order mutants.
- Find the killable second-order mutants that affect the du-pairs.

Table 3. No. of first order, no. of second order, and no. of du-pairs.

Tested Program	All 1st Order	Killable 1st Order	All 2nd Order	Killable 2nd Order	Du-Based Killable 2nd Order	Du-Pairs
Triangle	163	146	30,799	23,799	2459	102
Sort	120	104	15,960	11,547	2333	95
Stack	70	57	5146	3507	354	68
Reversed	98	78	10,444	6620	956	77
Clock	78	74	7638	7036	109	64
Quotient	156	140	27,184	21,338	2597	89
Product	117	101	17,018	12,321	759	71
Area	112	88	15,555	8656	405	64
Dist	83	73	9207	6774	274	64
Middle	74	70	6876	5458	319	70
Total	1071	931	145,827	107,056	10,565	764

4.4. Comparison Hypotheses

The mutation and data flow testing criteria will be compared according to the two relationships “ProbBetter” [16] and “ProbSubsumes” [3] that have been used to compare other testing criteria. In this comparison, the following hypotheses are formulated:

Hypothesis 1 (H1). *Second-order mutation testing “ProbSubsumes” all du-pairs data flow testing.*

Hypothesis 2 (H2). *All du-pairs data flow testing “ProbSubsumes” second-order mutation testing.*

Hypothesis 3 (H3). *Second-order mutation testing “ProbBetter” all du-pairs data flow testing.*

Hypothesis 4 (H4). *All du-pairs data flow testing “ProbBetter” second-order mutation testing.*

4.5. Experimental Results

4.5.1. Coverage Cost

For each subject program, the genetic algorithm finds a test cover to execute all du-pairs and another test cover to execute the du-based killable second-order mutants. The genetic algorithm has been applied many times to obtain more accurate results. The genetic algorithm has been applied three times on each subject program to find an adequate cover for the du-based mutants. In the same way, the genetic algorithm has been run on each subject program three times to find a sufficient covering for all du-pairs. The average number of the obtained test cases was computed for each subject program (given in Table 4). Table 5 gives the statistical results of the *t*-Test for the test-covers of the two criteria.

The data given in Table 4 show that all du-pairs testing criterion needs 20.3 test cases (on average) to cover all the set of du-pairs for each tested program. Furthermore, the second-order mutation testing criterion requires 59.1 test cases (on average) to kill all second-order mutants for each tested program. The statistical results of the *t*-Test (given in Table 5) show that the size of the test cover of all du-pairs criteria is significantly smaller than the size of the test cover of the second-order mutation criterion.

Table 4. Du-pairs and second-order mutants cover size.

Tested Program	Cover Size of 2nd Order Mutants	Cover Size of All Du-Pairs
Triangle	71.3	21.3
Sort	18.3	6.7
Stack	49.3	21.0
Reversed	34.7	15.3
Clock	101.3	32.7
Quotient	34.3	16.3
Product	79.7	21.3
Area	89.3	30.7
Dist	80.7	24.7
Middle	32.3	12.7
Average	59.1	20.3

Table 5. Test: paired two samples for means of cover size.

	2nd-Order Mutant	Du-Pairs
Mean	59.13	20.27
Variance	824.4	62.96
$P(T \leq t)$ two-tail	2.77×10^{-4}	

4.5.2. Coverage Adequacy

To investigate the "ProbSubsumes" relationship between the two testing criteria, each tested program is executed using the adequate test set concerning the second-order mutation testing criterion; and the adequacy concerning the all du-pairs testing criterion is measured. In addition, each tested program is executed using the adequate test set concerning the all du-pairs testing criterion; and the adequacy concerning the second-order mutation testing criterion is measured.

The mutants' coverage of the all du-pairs adequate test cover is given in Table 6, and the du-pairs coverage of the second-order mutants adequate test cover is given in Table 7. The coverage ratio for each of the three test suites is presented, as well as the average mutation and data flow coverages. Table 8 shows the statistical comparison between the obtained coverages and their significant difference. Figure 3 shows the mutual coverage of all du-pairs and second-order mutation criteria.

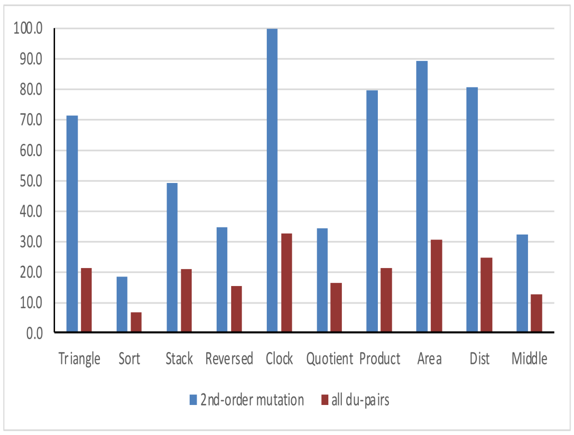


Figure 3. Mutual coverage of all du-pairs and second-order mutation.

Table 6. Second-order mutant coverage ratio using all du-pairs cover.

Tested Program	Suite1	Suite2	Suite3	Average
Triangle	98%	100%	98%	98.7%
Sort	96%	96%	96%	95.6%
Stack	89%	100%	89%	92.6%
Reversed	100%	100%	100%	100%
Clock	100%	100%	100%	100%
Quotient	100%	100%	100%	100%
Product	100%	100%	100%	100%
Area	100%	100%	100%	100%
Dist	100%	100%	100%	100%
Middle	100%	100%	100%	100%
Average	98.3%	99.6%	98.3%	98.7%

Table 7. All du-pairs coverage ratio using second-order mutant cover.

Tested Program	Suite1	Suite2	Suite3	Average
Triangle	80%	82%	80%	81%
Sort	85%	85%	85%	85%
Stack	89%	85%	85%	86%
Reversed	75%	80%	75%	77%
Clock	80%	80%	80%	80%
Quotient	70%	73%	70%	71%
Product	78%	80%	75%	78%
Area	80%	83%	88%	84%
Dist	79%	90%	79%	83%
Middle	85%	85%	85%	85%
Average	80.1%	82.3%	80.2%	80.9%

Table 8. *t*-Test: paired two sample for means of coverage ratio.

	Second-Order Mutant	Du-Pairs
Mean	98.7%	80.9%
Variance	0.1%	0.2%
P(T ≤ t) two-tail	1.11×10^{-5}	

The mean mutation coverage ratio of the all du-pairs adequate test cover is 80.9%, and the mean data flow coverage ratio of the second-order mutant adequate test cover is 98.7%. Therefore, one can accept H1 and reject H2. Consequently, second-order mutation testing "ProbSubsumes" the all du-pairs data flow testing.

4.5.3. Failure Detection Efficiency

A set of faults is seeded into each tested program to investigate the "ProbBetter" [16] relationship between the two testing criteria. Each program is executed using the adequate test suites of the second-order mutation and all du-pairs criteria. Then, the failure detection efficiency of each test suite is estimated. The number of faults seeded into each program is given in the second column of Table 9, while the third and fourth columns give the failure detection ratio for the test suite of the two criteria. The total number of faults seeded into the tested programs is 10,565 faults (1056 faults on average, 2459 faults on maximum, and 109 on minimum).

Table 9. Failure detection ratio of second-order mutation and all du-pairs test suites.

Tested Program	No. of Faults	Failure Detection Efficiency	
		2nd-Order Mutation	Du-Pairs
Triangle	2459	98%	91%
Sort	2333	96%	84%
Stack	354	89%	72%
Reversed	956	100%	82%
Clock	109	100%	75%
Quotient	2597	100%	92%
Product	759	100%	93%
Area	405	100%	86%
Dist	274	100%	89%
Middle	319	100%	92%
	Total = 10,565	Average = 98%	Average = 86%

The failure detection efficiency of the second-order mutation is 98% (on average), while the failure detection efficiency of the all du-pairs is 86% (on average). In addition, the second-order mutation criterion detects all the seeded faults in 7 out of 10 tested programs, while the all du-pairs criterion failed to detect 100% of faults in any tested program. Figure 4 shows the failure detection efficiency of the two criteria.

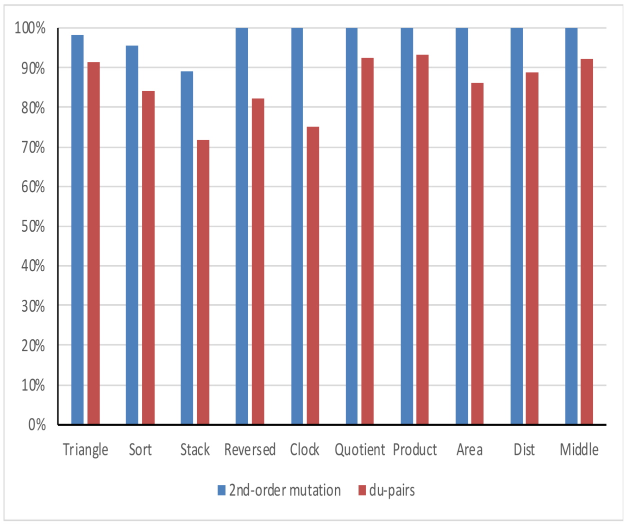


Figure 4. Failure detection efficiency.

Table 10 shows the statistical comparison between the obtained failure detection efficiency and their significant difference. According to the results of the *t*-Test, the *p*-value (9.195×10^{-5}) is less than 0.05; therefore, the failure detection efficiency of the mutation (98%) is significantly better than the failure detection efficiency of data flow (86%). Therefore, one can accept H3 and reject H4. Consequently, second-order mutation testing is "ProbBetter" than the all du-pairs data flow testing.

Table 10. *t*-Test: paired two sample for means of failure detection ratio.

2nd-Order Mutant		Du-Pairs
Mean	98.3%	85.7%
Variance	0.1%	0.6%
P(T $\leq$ t) two-tail	9.195×10^{-5}	

4.5.4. Threats to Validity

(1) External validity

The key external threat to validity is the subject programs are small-sized programs and are not large enough to argue that these programs are sufficiently representative of the overall population of programs. Although these programs are of small size, these programs have been used in several previous experimental studies, and they have identical constructions as the large-sized programs. Therefore, the suggested method has the capability to address large-sized or real programs.

(2) Internal validity

The key internal threat to validity is the setup of the parameters of the genetic algorithm. We used a subset of the tested programs to determine these parameters.

5. Conclusions and Future Work

Data flow and Higher-Order Mutation are white-box testing methodologies, and these two methodologies are the most popular and successful white-box testing techniques. This paper presented an empirical comparison of all def-uses data flow and second-order mutation criteria. This compassion investigated the subsumption relation between these two criteria and evaluated the effectiveness of test data developed for each. The results showed that the mean mutation coverage ratio of the all du-pairs adequate test cover is 80.9%, and the mean data flow coverage ratio of the second-order mutant adequate test cover is 98.7%. Consequently, second-order mutation testing "ProbSubsumes" the all du-pairs data flow testing. The failure detection efficiency of the mutation (98%) is significantly better than the failure detection efficiency of data flow (86%). Consequently, second-order mutation testing is "ProbBetter" than the all du-pairs data flow testing. In contrast, the size of the test suite of second-order mutation is bigger than the size of the test suite of all du-pairs. The results showed that the change in the parameters of the genetic algorithm affects the obtained ratio. In future work, more testing criteria will be compared with Higher-Order Mutation testing.

Author Contributions: Conceptualization, R.A.; Methodology, E.H.A.-E.; Software, E.H.A.-E.; Validation, R.A.; Formal analysis, R.A.; Investigation, R.A.; Data curation, E.H.A.-E.; Writing—original draft, E.H.A.-E.; Writing—review & editing, R.A.; Supervision, R.A.; Project administration, E.H.A.-E. All authors have read and agreed to the published version of the manuscript.

Funding: This research study is funded by the grant no. CSCR-2022-11-1709 from the Deanship of Scientific Research at Northern Border University, Arar, K.S.A.

Data Availability Statement: All result data produced from applying testing techniques by the authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. White, L.J. Software testing and verification. In *Advances in Computers*; Yovits, M.C., Ed.; Elsevier: Amsterdam, The Netherlands, 1987; Volume 26, pp. 335–390.
2. Offutt, A.J.; Pan, J.; Tewary, K.; Zhang, T. An Experimental Evaluation of Data Flow and Mutation Testing. *J. Softw. Pract. Exp.* **1996**, *26*, 165–176. [CrossRef]
3. Mathur, A.P.; Wong, W.E. An empirical comparison of data flow and mutation-based adequacy criteria. *Softw. Test. Verif. Reliab.* **1994**, *4*, 9–31. [CrossRef]

4. Offutt, A.J.; Voas, J.M. *Subsumption of Condition Coverage Techniques by Mutation Testing*; Technical Report ISSE-TR-96-01; Information and Software Systems Engineering George Mason University: Fairfax, VA, USA, 1996.
5. Frankl, P.G.; Weiss, S.N.; Hu, C. All-uses vs. mutation testing: An experimental comparison of effectiveness. *J. Syst. Softw.* **1997**, *38*, 235–253. [CrossRef]
6. Kakarla, S.; Momotaz, S.; Namin, A.S. An Evaluation of Mutation and Data-Flow Testing: A Meta-analysis. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011.
7. Bluemke, I.; Kulesza, K. A Comparison of Dataflow and Mutation Testing of Java Methods. In *Dependable Computer Systems*; Advances in Intelligent and Soft Computing; Zamojski, W., Kacprzyk, J., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011.
8. Mathur, A.P.; Wong, W.E. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study. *Softw. Qual. J.* **1994**, *4*, 69–83.
9. Tewary, K.; Harrold, M.J. Fault modeling using the program dependence graph. In Proceedings of the Fifth International Symposium on Software Reliability Engineering, Monterey, CA, USA, 6–9 November 1994.
10. Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers, Principles, Techniques, and Tools*; Addison-Wesley Publishing Company: Boston, MA, USA, 1986.
11. Frankl, P.G.; Weyuker, E.J. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.* **1988**, *14*, 1483–1498. [CrossRef]
12. Jia, Y.; Harman, M. Higher order mutation testing. *Inf. Softw. Technol.* **2009**, *51*, 1379–1393. [CrossRef]
13. Ghiduk, A.S.; Girgis, M.R.; Shehata, M.H. Higher-order mutation testing: A systematic literature review. *Comput. Sci. Rev. J.* **2017**, *25*, 9–48. [CrossRef]
14. DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer* **1978**, *11*, 4–41. [CrossRef]
15. Hamlet, R.G. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* **1977**, *3*, 279–290. [CrossRef]
16. Weyuker, E.J.; Weiss, S.N.; Hamlet, R.G. Comparison of program testing strategies. In Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification, Victoria, BC, Canada, 8–10 October 1991.
17. Michalewicz, Z. *Genetic algorithms + Data Structures = Evolution Programs*, 3rd ed.; Springer: Berlin/Heidelberg, Germany, 1998.
18. Harman, M.; Jia, Y.; Langdon, B. Strong higher order mutation-based test data generation. In Proceedings of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11), Szeged, Hungary, 5–9 September 2011.
19. Dang, X.; Gong, D.; Yao, X.; Tian, T.; Liu, H. Enhancement of Mutation Testing via Fuzzy Clustering and Multi-population Genetic Algorithm. *IEEE Trans. Softw. Eng.* **2021**, *48*, 2141–2156. [CrossRef]
20. Derezińska, A.; Szustek, A. *CREAM—A System for Object-Oriented Mutation of C# Programs*; Warsaw University of Technology: Warszawa, Poland, 2007.
21. Derezińska, A. *Advanced Mutation Operators Applicable in C# Programs*; Warsaw University of Technology: Warszawa, Poland, 2005.
22. Derezińska, A. Quality assessment of mutation operators dedicated for C# programs. In Proceedings of the 6th International Conference on Quality Software (QSIC'06), Beijing, China, 27–28 October 2006.