

Received October 9, 2019, accepted November 1, 2019, date of publication November 7, 2019, date of current version November 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2952246

Employing Dynamic Symbolic Execution for Equivalent Mutant Detection

AHMED S. GHIDUK^{1,2}, MOHEB R. GIRGIS³, AND MARWA H. SHEHATA²

¹College of Computers and IT, Taif University, Taif 21974, Saudi Arabia

²Department of Math and CS, Faculty of Science, Beni-Suef University, Beni Suef 62521, Egypt

³Department of Computer Science, Faculty of Science, Minia University, El-Minia 61519, Egypt

Corresponding author: Ahmed S. Ghiduk (asaghiduk@yahoo.com)

ABSTRACT Equivalent mutants (EM) issue is a key challenge in mutation testing. Many methods were applied for detecting and reducing the equivalent mutants. These methods are classified into four classes: equivalent mutant detection, avoiding the generation of equivalent mutants, higher-order equivalent mutants, and suggesting equivalent mutants. Higher-order mutation testing (HOMT) is considered the strongest employed technique in avoiding the generation of equivalent mutants and reduction of their number. In this paper, a combination of HOMT especially second-order mutation testing (SOMT) and dynamic symbolic execution (DSE) techniques are applied for the automatic detection and reduction of the equivalent second-order mutants. First, SOMT is used to reduce the number of equivalent mutants. Second, DSE technique is applied to classify the SOMs and detect EM. To assess the efficiency of the proposed technique, it is applied to some subject programs and the results of this technique are compared to the manual results and those of related works. The results showed that the proposed algorithm is more effective in detecting and reducing the number of EM. It detects 94% from the equivalent mutants that have manually analyzed. This percentage is a high percentage comparing with previous studies. Besides, the DEM-DSE technique detects 100% of equivalent mutants for 9 of the 14 subject programs.

INDEX TERMS Higher-order mutation testing, dynamic symbolic execution, equivalent mutants.

I. INTRODUCTION


Mutation testing (MT) depends on the idea of generating one or more mutated versions for the source program to use them in estimating the quality of the test suite. Mutation testing was founded in the 1970s by DeMillo *et al.* [1] and Hamlet [2]. It is classified into two types: first-order mutation testing and higher-order mutation testing [3], [4].

The mutant (faulty program) may be a first-order mutant (FOM) or higher-order mutant (HOM). A FOM is generated by creating only one error in the source program while a HOM is generated by creating two or more errors in the source program. The source program and the mutated versions are executed by a test suite. If the output of the source program is dissimilar to the result of the faulty version (mutant), this mutant is called a “Killed” mutant. Otherwise, it is called an “Alive” mutant which may be killed or equivalent one. Error creation is made by applying the mutation operators to the source program. Some of the

mutation operators produce mutants semantically similar to the source program which called equivalent mutants. These mutants cannot be killed and a lot of human effort is required to detect these mutants.

Although MT is a highly effective technique to evaluate the quality of the test cases, it suffers from some problems such as equivalent mutant problem, realism problem, and generating a large number of mutants. One of the most difficult problems in mutation testing is the EM problem. EM is a mutant that cannot be killed by any test case. In recent years, there are many proposed techniques are applied to detect and reduce the number of equivalent mutants. Madeyski *et al.* [5] classified EM problem techniques into three groups:

- Techniques for the detection of equivalent mutants such as compiler optimization [6], [7], mathematical constraints [8], program slicing [9], and laser model checker [10].
- Techniques to reduce the number of equivalent mutants such as selective mutation [11], program dependence analysis [12], co-evolutionary [13], and higher-order mutation testing [3], [5], [14].

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana .

- Techniques to suggest the equivalent mutants such as Bayesian-learning based guidelines [15], the impact coverage [16], and dynamic invariants impact [17].

SOMT approaches [3], [4] and HOMT approaches [5] are considered significant solutions for the mutation testing problems. SOMT depends on generating second-order mutants (SOMs) that are created by adding two faults into the source code. HOMT depends on generating higher-order mutants (HOMs) that are created by seeding two or more errors in the source code. SOMT techniques specially and HOMT techniques generally [3], [4], [18] are considered the most significant techniques for solving the EM problem. These techniques combine two or more FOMs to generate SOM or HOM which reduces the number of mutants by 50% or more and also decreases the number of EM. In addition, Gong *et al.* [41] proposed a technique for reducing the number of mutants by considering the dominance relation between mutant branches. The results showed that this technique reduces 80% of mutants on average.

Bearing these ideas in mind, the current study aims to solve the EM problem through detecting and reducing the number of second-order EM. For this reason, this paper proposes a new technique using the SOM testing techniques [18] and the DSE techniques [19], [20]. Firstly, SOMT is used to reduce the number of mutants that helps to reduce the number of EM. Secondly, the idea of the DSE is used to execute the tested program and its mutants with initial random test inputs and using symbolic values for these inputs in parallel to collect some information for each of the tested programs and its mutants. Such information is represented in the variables operators, executed code lines, constraint paths, and program output. This information helps to classify the mutants and detect the equivalent mutants.

Detecting equivalent mutants in this work depends on executing the source code and its mutants concretely and symbolically at the same time (which called DSE) and then compare between them through the information that collected about them during that execution. This technique considers the mutant surely “Killed” when the information is different and the mutant is reached. Otherwise, the mutant may be called “Equivalent” or “Strong”. To determine that the mutant is surely “equivalent” or surely “strong”, the technique re-executes the source code and the mutant with a large number of test data.

Testing the software application may be executed concretely by using random input values or symbolically by using symbolic values that generate constraint paths which are solved to generate new concrete values. The DSE is a hybrid software verification technique where executes the symbolic execution along concrete execution. DSE testing executes a program starting with some random or given input, collect symbolic constraints using symbolic values along the execution on inputs at conditional statements, and then uses a constraint solver to generate the new test inputs. Most of the previous EM detected problem techniques [21], [22]

TABLE 1. An example of equivalent FOM.

Original Program p	Mutated Program p' Equivalent FOM
F(float a, float b){ float result; if(a < b) result = a + b; else result = a - b; return result ;}	F(float a, float b){ float result; if(a < b) result = a + b++; else result = a - b; return result ;}

TABLE 2. An example of equivalent SOM.

Source Program p	Mutated Program p'		
	FOM1	FOM2	Equivalent SOM
F(float a, float b) { float result; if(a < b) result = a + b; else result = a - b; return result ;}	F(float a, float b) { float result; if(a < b) result =a+b++; else result = a - b; return result ;}	F(float a, float b) { float result; if(a < b) result =a+b; else result =a - b--; return result ;}	F(float a, float b) { float result; if(a < b) result =a+b++; else result =a - b--; return result ;}

can detect EM by percentage 45% to 70%. On the contrary, the proposed technique can detect about 94% from EM that analyzed manually.

The rest of this paper is organized as follows. Section II presents related work. Section III presents the main idea of DSE. Section IV presents the proposed technique for detecting equivalent mutants. The discussion the setup of the experiments and their results are presented in Section V. The conclusion of the paper is presented in Section VI.

II. RELATED WORK

The equivalent mutant (EM) is created once the behavior of the mutant is identical to the source program and there is no difference between them. Tables 1 and 2 give examples of equivalent FOMs and SOMs. Detecting EM needs high-quality test cases and more human effort. The quality of any EM technique depends on its ability for detecting most EM that analyzed manually. The greater the number of EM detected, the greater the percentage of mutation score.

The process of detecting EM manually is very expensive and takes a long time. Therefore, various approaches have been proposed to detect EM automatically. In the last twenty years, the number of studies that were proposed to solve the EM problem became increasing. These approaches are classified into three classes as mentioned above. The most effective techniques for overcoming EM problem are SOMT [5], [23] in particular and HOMT [24], [25] in general. SOMT or HOMT techniques could decrease the number of mutants and reduce the number of EM.

In 1979, a study proposed by Baldwin and Sayward [6] detects EM by using a compiler optimization technique. This approach [6] presented different types of compiler optimization rules to detect EM. This approach detected that 10% of all mutants were EM for 15 tested programs.

Offutt and Pan [8], [26], [42] used constraint solving for detecting EM. These approaches [8], [26], [42] formulated the EM problem as a constraint problem, analyzed the path condition of a mutant, and solved the constraint to detect EM. If no solution is found then EM is detected. The Empirical results of these approaches showed that using constraint technique able to detect EM in percentage 47.63% for 11 subject programs. The same technique was used by Nica and Wotawa [43] to detect the EM in percentage 40%.

The program slicing technique was used to detect EM [9]. The program slicing technique [9] isolated the components of a program. These components are the computation of a single variable or set of variables. Detecting EM is achieved by the generation of a sliced program.

The study proposed by Grun et al. [16] determined the impact of mutants during the execution to detect EM. The impact of a mutant was measured using code coverage and was represented as the difference in the program behavior between the mutant and the original program. Schuler et al. [27] used the impact coverage to distinguish between the equivalent mutant and nonequivalent mutant. This approach was applied to seven tested programs to test its efficiency. The percentage of EM ranges from 25% to 70%.

In 2012, a new dynamic method called isolating equivalent mutant [23] was proposed to classify mutants to possibly killable or possibly equivalent mutants. This technique based on using SOM testing to classify the FOMs as possibly killable or not.

Kintis [37] and Orzeszyna [38] discussed in detail the proposed methods to handle the equivalent mutant problem.

III. DYNAMIC SYMBOLIC EXECUTION

So far, there are many techniques used to test software applications such as random testing techniques [28], [29], symbolic execution techniques [19], and search-based techniques [33]–[36], [39], [40]. The DSE [20] is considered a combination of random testing and symbolic execution techniques [19], [29]. In random execution, actual values are used to execute the program. While in symbolic execution, symbolic values are used instead of actual values to execute the program. In recent years, the symbolic execution [19] and DSE [20] techniques are used in software testing for generating high coverage test suites and for detecting the errors in software applications. The DSE [20] is considered a combination of symbolic and concrete execution where executes the program with initial random inputs and using symbolic values for these inputs in parallel to collect the symbolic constraints after that DSE solves these constraints to generate new test input.

Figures 1 and 2 provide examples of using random testing, symbolic execution, and DSE to test a simple java source code. Figure 1 gives a Java source code where the method F takes two inputs H and G and calculates the output K. Figure 2 describes how can each of random, symbolic, and DSE techniques test the source code given in Figure 1.

```

F(int G, int H) {
    int K;
    if (G < H) {
        K = G * H;
        break; }
    else {
        K = G / H;
        break; }
}
    
```

FIGURE 1. Simple java source.

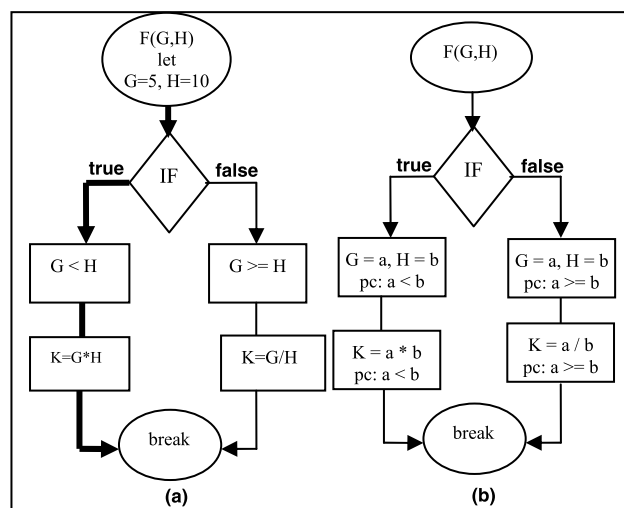


FIGURE 2. Random and symbolic execution.

Suppose the created random values for F(G, H) are G = 5, and H = 10; then random testing technique will cover the true branch as given in Figure 2(a) and the output will be k = 50.

Suppose the symbolic values for F(G, H) are G = a, and H = b; then symbolic execution technique will find to constraints: PC1: a < b for the true branch and PC2: a >= b for the false branch as given in Figure 2(b). These constraints can be solved by a constraint solver to find the actual values.

For applying DSE, the program is executed concretely and symbolically as given in Figure 2. Then, the symbolic execution generates the path constraint a < b for the path that was executed randomly. Consequently, DSE negates this path constraint and solves the new path constraint a >= b to generate new test input values.

IV. OUR PROPOSED TECHNIQUE (DEM-DSE)

This section describes the proposed technique for Detecting Equivalent Mutants using Dynamic Symbolic Execution (DEM-DSE). The technique decreases the number of EM, detects the EM, and classifies the mutants into two categories: killed mutants (KM) and EM. The DEM-DSE technique depends on the combination of the SOM technique and DSE technique. The proposed DEM-DSE technique uses MuClipse [30] tool to generate FOMs, our previous SOM testing technique given in [18] which uses SCWR Algorithm

```

BEGIN
1. Get the original tested program P
2. Applying MuClipse tool on P to generate FOMs
3. Applying SCWR algorithm on FOMs to generate SOMs list
4. Return SOMs
END

```

FIGURE 3. Gen-SOMs. Algorithm.

```

BEGIN
1. Get original program P
2. Let info contains covered operators, constraint paths, code
   lines, the output results and the Reachability of mutant
3. Let SM list is SOMs list
4. Let TS set is random test inputs
5. Call Gen_SOMs Algorithm to get SM list
6. Initialize Equivalent SOMs (ESM) list to NULL
7. Initialize Killed SOMs (KSM) list to NULL
8. Instrument P to InstP (using symbolic values)
9. FOR each M in SM list
10. Instrument M to InstM (using symbolic values)
11. Save InstM to InstSM list
12. END FOR
13. FOR each InstM in InstSM list
14. Execute InstM using TS to get info_M
15. Execute InstP using TS to get info_P
16. Compare info_M with info_P
17. IF similar THEN Add M to ESM list
18. ELSE
19. Add M to KSM list
20. END IF
21. END FOR
END

```

FIGURE 4. DEM-DSE Algorithm.

to generate the SOMs, and applies the idea of DSE [20] to detect the equivalent mutants.

DEM-DSE technique is performed as follows.

Firstly, Gen-SOMs Algorithm given in Figure 3 generates the FOMs for the java tested program using MuClipse tool and then applies the SCWR Algorithm [18] to generate its SOMs. It receives as input the source program and it gives as output its SOMs. SCWR Algorithm [18] has two features; it reduces the number of the mutants to about 60% comparing to FOMs and also reduces the number of EM mutants to less than half compared to FOMs.

Secondly, DEM-DSE algorithm given in Figure 4 which receives as input the source program and its SOMs, then instruments the source program and its SOMs using symbolic values.

Thirdly, the instrumented program and the instrumented SOMs are executed with initial test inputs which are created randomly to collect some information for each of them. This information is represented in the following: the covered operators, the path constraints, the executed code lines, the output of the program or SOM, and reachability for each mutant [31]. To classify the SOMs, the information of the source program and the information of each SOM are compared correspondingly.

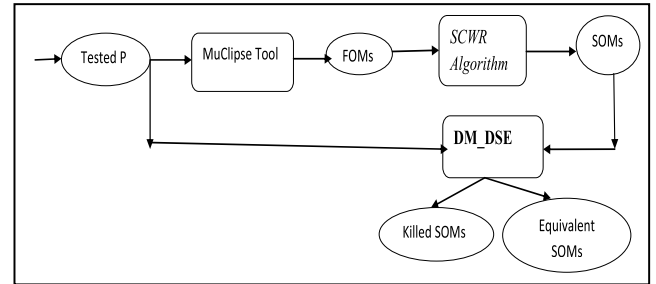


FIGURE 5. DEM-DSE architecture.

Finally, the DEM-DSE technique classifies the mutants according to the following set of rules.

- Rule 1: if M is a killed mutant according to operators only then M is an equivalent mutant (EM).
- Rule 2: if M is a killed mutant according to all conditions (output, path, operators, and constraints) then M is a naïve mutant.
- Rule 3: if M is a killed mutant according to output only or output and any other condition then M is a normal mutant.
- Rule 4: otherwise M is a strong mutant.

A mutant is operators, constrains, executed path, or output based-killed if the executed operators, executed path, executed constrains, or output of the source program is different from the corresponding of the mutant.

The mutant is equivalent if it is killed by the operators only (Rule 1) (e.g., FOM#1 and SOM#1 given in Tables 6(A) and 6(B), respectively). The mutant is naïve if it is killed by all the four methods (operators, constrains, executed path, and output) (Rule 2) (e.g., FOM#33 and SOM#2 given in Tables 6(A) and 6(B), respectively). The mutant is normal if it is killed by output with or without another method (Rule 3) (e.g., FOM#3 and SOM#3 given in Tables 6(A) and VI(B), respectively). Otherwise, the killed mutant is called a strong mutant (Rule 4) (e.g., FOM#2 (killed by operators and constraints). Although the classification based only on the rules (1-4), to be sure that an “alive” mutant is surely “strong” or surely “equivalent, we execute this mutant with more and more test cases that are randomly created. These test cases can be generated using any other technique such as evolutionary algorithms [39] or genetic algorithms [34], [36], [40].

The details of the proposed technique and the experimental study are discussed in the following section.

V. EXPERIMENTAL SETUP AND RESULTS

This section gives the details of the conducted experiment to assess the proposed DEM-DSE technique for detecting equivalent first and second-order mutants (EFOMs and ESOMs).

Figure 5 shows the architecture of the DEM-DSE technique, which consists of three modules: FOMs generation module which applies MuClipse [30], SOMs generation

TABLE 3. Subject programs.

Subject Programs	Reference	Scale	#Test Cases
P#1. Triangle	[3,18]	1 C, 37 LOC	50
P#2. CalDay	[30,18]	1C, 55LOC	43
P#3. Remainder	[30]	1C, 46 LOC	35
P#4. Maximum	[30]	1C, 70 LOC	25
p#5. Mid	[18]	1C, 50 LOC	10
P#6. CPrime	[18]	1C, 30 LOC	12
P#7. Power	[30]	1 C, 27 LOC	35
P#8. Bub	[24]	1C, 69 LOC	18
P#9. Bisect	[24]	1C, 58 LOC	18
P#10.Doubly-Linked-List	[30]	1 C, 277 LOC	48
P#11. Joda-time	[31]	157 C, 25,905 LOC	20
P#12. Pamvotis	[31]	26 C, 5,149 LOC	100
P# 13. Xstream	[31]	209 C, 16,791 LOC	58
P#13-1.Xstream Class			
P13-2. QuickWriter Class			
P#14.Commons	[31]	100 C, 19,583 LOC	43
P#14-1.WordUtils Class			
P#14-2. NumberUtils Class			

module which applies SCWR Algorithm [18], and SOMs classification module which applies DSE [20].

A. EXPERIMENTAL SETUP

1) SUBJECT PROGRAMS

In this experiment, a set of Java programs was selected from the previous studies for conducting the experiment to evaluate the proposed technique. This set contains two forms of programs, small-sized programs which consist of one class such as Triangle, Mid, Remainder, CalDay, Maxim, CPrime, Bub, Bisect, Power, and Doubly-Linked-List; and large-sized programs which consist of more than one class such as Commons from which WordUtils and NumberUtils classes are selected, Joda-time from which DateTime class is selected, Pamvotis from which Simulator is selected, and Xstream from which Xstream and Quick Writer are selected.

Table 3 presents the subject programs details: the “Subject Program” column represents the program title; the “Reference” column represents the studies that used these set of programs; the “Scale” column represents the number of classes, and code lines in each subject program, and “# Test Cases” represent the number of test cases used to test the programs. Portion of the test cases is created randomly and the others are created using the DSE technique. Whereas, the initial test cases are created randomly and the rest test cases are created by solving the collected constraints.

2) MUTATION TOOL

In the experiment, the MuClipse tool [30] is applied. MuClipse is a mutation testing tool for Java programs. MuClipse tool depends basically on the implementation of the MuJava tool [32] that is a very popular mutation testing tool. MuClipse tool helps the testers in generating the mutants and running the tests against these mutants.

MuClipse tool [30] uses two mutation operator types to generate the FOMs. The used mutation operators are Class- Level and Traditional-Level operators. The tradi-

TABLE 4. Traditional- level operators.

Category	Operator	Description
Arithmetic	AOR _B	Arithmetic Operator Replacement (binary)
	AOR _U	Arithmetic Operator Replacement (unary)
	AOR _S	Arithmetic Operator Replacement (short-cut)
	AOI _U	Arithmetic Operator Insertion (unary)
	AOI _S	Arithmetic Operator Insertion (short-cut)
Relational	AOD _U	Arithmetic Operator Deletion(unary)
	AOD _S	Arithmetic Operator Deletion(short-cut)
	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
	LOR	Logical Operator Replacement
Logical	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASR _S	Assignment Operator Replacement(short-cut)

tional operators only are used in the experiment because the proposed algorithm compares the operators that were used in the tested program and its mutants that are represented in six types of Traditional-Level operators. The Traditional-Level operators’ types are (see Table 4) as follows: Arithmetic operator, Relational operators, Conditional operators, Shift operators, Logical operators, and Assignment operators.

3) EXPERIMENT PROCEDURE

The experiment was performed as follows:

- 1) Set the configuration of mutation operators feature of MuClipse to Traditional-Level operators; then apply it on the tested program to generate FOMs.
- 2) Apply the SCWR Algorithm [18] on the FOMs to create SOMs.
- 3) Run the DEM-DSE algorithm to classify the mutants and detect EM automatically considering the four proposed rules (Rules 1, 2, 3, and 4).
- 4) To assess the effectiveness of the proposed technique, the results are compared to the manual results and also to the results of the related work [21].

In the following, an example is given to illustrate the experiment procedure and the algorithm on each of FOMs and SOMs.

First, let the tested program is a Java program which reads three integers and finds the maximum value as given in Figure 6(a). This program needs three inputs to be executed.

Second, the MuClipse tool is applied to generate FOMs as given in Figure 6(b) and then generate SOMs by the SCWR Algorithm [18] as given in Figure 6(c).

Third, the DEM-DSE algorithm is executed on the tested program example, FOMs and SOMs. To classify the mutants to “alive” or “killed” the DEM-DSE technique collects some information which are variable operators, executed code lines, constraint paths, program outputs for each input that satisfies the reachability condition [31]. The technique classifies “killed” mutant into “naïve”, or “normal”, and the “alive” mutant into “equivalent”, or “strong”.

```

public class maximum {
    public static int max( int a, int b, int c ) {
        1. int max = a;
        try {
            2. if (b > max) {
                3. max = b; }
            4. if (c > max) {
                5. max = c; }
        }
        catch(java.lang.ArithmeticException arithmeticexception )
            { System.out.print( "error" ); }
        6. return max;
    }
}
    
```

(A). Tested program example.

```

public class maximum
{
    public static int max( int a, int b, int c )
    {
        1. int max = a++;
        try {
            2. if (b > max) {
                3. max = b;
            }
            4. if (c > max) {
                5. max = c;
            }
        }
        catch(java.lang.ArithmeticException arithmeticexception ) {
            System.out.print( "error" ); }
        6. return max;
    }
}
    
```

(B) . FOM.

```

public class maximum
{
    public static int max( int a, int b, int c )
    {
        1. int max = a++;
        try {
            2. if (b > max--) {
                3. max = b;
            }
            4. if (c > max) {
                5. max = c;
            }
        }
        catch(java.lang.ArithmeticException arithmeticexception ) {
            System.out.print( "error" ); }
        6. return max;
    }
}
    
```

(C) . SOM.

FIGURE 6. (A) Tested program example. (B).FOM. (C).SOM.

This classification depends on the collected information during the execution of the tested program, FOM, and SOM. For example, let the inputs of the tested program example given in Figure 6(a) are a = 7, b = 9, and c = 5.

After the execution of tested program example, FOM, and SOM given in Figures 6(a), (b) and (c), respectively, the collected information for the tested program is:

Operators: {>; > }
 Constraints: {b > max; c <= max }

Code lines: {1, 2, 3, 4, 6}
 Output: 9

The collected information for the FOM is:

Operators: {++; >; > }
 Constraints: {b > max; c <= max }
 Code lines: {1, 2, 3, 4, 6}
 Output: 9

The collected information for the SOM is:

Operators: {++; >; >; - }
 Constraints: {b > max-; c <= max }
 Code lines: {1, 2, 3, 4, 6}
 Output: 9

Then, the technique applies DSE and solves the constraints b <= max and c > max where max = a (i.e., b <= a and c >= a). After solving these constraints, the new inputs can be a = 10, b = 8, and c = 12. After the execution of tested program example, FOM, and SOM given in Figures 6(a), (b) and (c), respectively, using the new inputs, the collected information for the tested program is:

Operators: {>; > }
 Constraints: {b <= max; c > max }
 Code lines: {1, 2, 4, 5, 6}
 Output: 12

The collected information for the FOM is:

Operators: {++; >; > }
 Constraints: {b <= max; c > max }
 Code lines: {1, 2, 4, 5, 6}
 Output: 12

The collected information for the SOM is:

Operators: {++; >; >; - }
 Constraints: {b <= max-; c > max }
 Code lines: {1, 2, 4, 5, 6}
 Output: 12

According to the output, code lines, and constraint (post-decrement operator - in “b <= max-” and “b > max-” are neglected because they don’t change the constraint) FOM and SOM are alive mutants which may be equivalent mutants or strong ones. According to operators FOM and SOM are killed. Therefore, FOM and SOM are equivalent mutants. By the manual checking of these mutants, these FOM and SOM are equivalent mutants.

The results of the tested program example are reported in two tables: Tables 5(A) and 5(B) that present information comparison between the tested program example and its mutants using initial random data (R) and new data (N) generated by DSE. The mutant is called “killed” when its information is different from the tested program information and “alive” mutant otherwise. In addition, Tables 6(A) and 6 (B) present the mutants classification.

From Tables 5 and 6, the MuClipse generated 52 FOMs which are manually classified into 41 “killed” mutants and 11 “equivalent”. When applying our algorithm, we found 11 “equivalent” and 41 “killed” that are classified into “Naïve” and “Normal “. Also for SOMs, the number of SOMs is 26 which are classified manually into 2 “equivalent” and 24 “killed” mutants. After execution of our algorithm, these

TABLE 5. Information comparison between Tested program and its FOMs.

FOM	(A)			
	Operators R;N;Resultant	Path R;N;Resultant	Constraints R;N;Resultant	Output R;N;Resultant
FOM#1	k v k=k	a v a=a	a v a=a	a v a=a
FOM#2	a v k=k	a v a=a	a v k=k	a v a=a
FOM#3	k v k=k	a v a=a	a v a=a	k v a=k
FOM#4	a v k=k	a v a=a	a v a=a	a v k=k
FOM#5	k v a=k	a v a=a	a v a=a	a v a=a
FOM#6	k v a=k	a v a=a	a v a=a	a v a=a
FOM#7	k v a=k	a v a=a	k v a=k	k v a=k
FOM#8	k v k=k	a v a=a	a v k=k	a v k=k
FOM#9	k v a=k	a v a=a	k v a=k	k v a=k
FOM#10	a v k=k	a v a=a	a v k=k	a v k=k
FOM#11	k v a=k	a v a=a	k v a=k	a v k=k
FOM#12	k v k=k	a v a=a	a v a=a	a v a=a
FOM#13	k v a=k	a v a=a	k v a=k	a v k=k
FOM#14	k v a=k	a v a=a	k v a=k	a v k=k
FOM#15	k v a=k	a v a=a	k v a=k	a v k=k
FOM#16	a v k=k	a v a=a	a v a=a	a v k=k
FOM#17	a v k=k	a v a=a	a v a=a	a v k=k
FOM#18	k v a=k	a v a=a	a v a=a	a v a=a
FOM#19	a v k=k	a v a=a	a v a=a	a v a=a
FOM#20	k v k=k	a v a=a	a v a=a	a v a=a
FOM#21	k v k=k	a v a=a	a v a=a	a v a=a
FOM#22	k v a=k	a v a=a	k v a=k	a v a=a
FOM#23	a v k=k	a v a=a	a v k=k	a v a=a
FOM#24	k v a=k	a v a=a	k v a=k	a v a=a
FOM#25	a v k=k	a v a=a	a v k=k	a v a=a
FOM#26	k v a=k	a v a=a	k v a=k	a v k=k
FOM#27	a v k=k	a v a=a	a v k=k	k v a=k
FOM#28	a v k=k	a v a=a	a v k=k	a v a=a
FOM#29	k v k=k	a v a=a	a v a=a	a v a=a
FOM#30	k v k=k	a v a=a	a v a=a	a v k=k
FOM#31	a v k=k	a v a=a	a v a=a	a v k=k
FOM#32	k v k=k	a v a=a	a v a=a	k v k=k
FOM#33	a v k=k	k v k=k	k v k=k	k v k=k
FOM#34	k v k=k	k v k=k	k v k=k	k v k=k
FOM#35	k v k=k	a v a=a	a v k=k	a v a=a
FOM#36	k v k=k	k v k=k	k v k=k	k v a=k
FOM#37	k v a=k	a v a=a	k v a=k	a v a=a
FOM#38	a v k=k	a v a=a	a v a=a	a v k=k
FOM#39	k v k=k	k v k=k	k v k=k	a v k=k
FOM#40	a v k=k	a v a=a	a v k=k	a v a=a
FOM#41	a v k=k	a v a=a	a v a=a	a v k=k
FOM#42	k v k=k	a v a=a	a v a=a	k v k=k
FOM#43	a v k=k	a v a=a	a v a=a	a v a=a
FOM#44	k v k=k	a v k=k	k v k=k	a v k=k
FOM#45	k v k=k	k v k=k	k v k=k	a v k=k
FOM#46	k v k=k	k v k=k	k v k=k	a v k=k
FOM#47	a v k=k	a v k=k	a v k=k	a v k=k
FOM#48	k v k=k	a v k=k	k v k=k	a v a=a
FOM#49	a v k=k	a v a=a	a v a=a	a v a=a
FOM#50	k v k=k	k v k=k	k v k=k	k v k=k
FOM#51	k v k=k	k v k=k	k v k=k	k v k=k
FOM#52	a v k=k	a v k=k	a v k=k	a v k=k
No. alive	0	40	21	21
No.killed	52	12	31	31
Total	52	52	52	52

TABLE 6. (a). FOMs Classification. (b). SOMs classification.

FOM	(A)	
	Automatic classification	Manual classification
FOM#1	Equivalent	Equivalent
FOM#2	Strong	Killed
FOM#3	Normal	Killed
FOM#4	Normal	Killed
FOM#5	Equivalent	Equivalent
FOM#6	Equivalent	Equivalent
FOM#7	Normal	Killed
FOM#8	Normal	Killed
FOM#9	Normal	Killed
FOM#10	Normal	Killed
FOM#11	Normal	Killed
FOM#12	Equivalent	Equivalent
FOM#13	Normal	Killed
FOM#14	Normal	Killed
FOM#15	Normal	Killed
FOM#16	Normal	Killed
FOM#17	Normal	Killed
FOM#18	Equivalent	Equivalent
FOM#19	Equivalent	Equivalent
FOM#20	Equivalent	Equivalent
FOM#21	Equivalent	Equivalent
FOM#22	Strong	Killed
FOM#23	Strong	Killed
FOM#24	Strong	Killed
FOM#25	Strong	Killed
FOM#26	Normal	Killed
FOM#27	Normal	Killed
FOM#28	Strong	Killed
FOM#29	Equivalent	Equivalent
FOM#30	Normal	Killed
FOM#31	Normal	Killed
FOM#32	Normal	Killed
FOM#33	Naïve	Killed
FOM#34	Naïve	Killed
FOM#35	Strong	Killed
FOM#36	Naïve	Killed
FOM#37	Strong	Killed
FOM#38	Normal	Killed
FOM#39	Naïve	Killed
FOM#40	Strong	Killed
FOM#41	Normal	Killed
FOM#42	Normal	Killed
FOM#43	Equivalent	Equivalent
FOM#44	Naïve	Killed
FOM#45	Naïve	Killed
FOM#46	Naïve	Killed
FOM#47	Naïve	Killed
FOM#48	Strong	Killed
FOM#49	Equivalent	Equivalent
FOM#50	Naïve	Killed
FOM#51	Naïve	Killed
FOM#52	Naïve	Killed
Total of killed	K=Na+ S + No = 11+10+20=41	41
Total of Equivalent	11	11

SOM	(B)	
	Automatic classification	Manual classification
SOM#1	Equivalent	Equivalent
SOM #2	Naïve	Killed
SOM #3	Normal	Killed
SOM #4	Naïve	Killed
SOM #5	Naïve	Killed
SOM#6	Normal	Killed
SOM #7	Normal	Killed
SOM #8	Normal	Killed
SOM #9	Normal	Killed
SOM #10	Equivalent	Equivalent
SOM #11	Normal	Killed
SOM #12	Normal	Killed
SOM #13	Normal	Killed
SOM #14	Naïve	Killed
SOM #15	Normal	Killed
SOM #16	Normal	Killed
SOM #17	Naïve	Killed
SOM #18	Naïve	Killed
SOM #19	Normal	Killed
SOM #20	Naïve	Killed
SOM #21	Normal	Killed
SOM #22	Naïve	Killed
SOM #23	Naïve	Killed
SOM #24	Naïve	Killed
SOM #25	Naïve	Killed
SOM#26	Naïve	Killed
Total of killed	K=Nv+S+Nr= 12+0+12=24	24
Total of Equivalent	2	2
Total	26	26

TABLE 7. Number of FOMs and SOMs.

Tested Program	NO.FOMs	NO.SOMs	%Reduction ratio
P#1	108	54	50.0%
P#2	48	23	52.1%
P#3	84	41	51.2%
P#4	52	26	50.0%
P#5	94	42	55.3%
P#6	43	20	53.5%
P#7	52	26	50.0%
P#8	49	23	53.1%
P#9	89	35	60.7%
P#10	105	50	52.4%
P#11	560	280	50.0%
P#12	5885	2942	50.0%
P#13-1	345	170	50.7%
P#13-2	154	75	51.3%
P#14-1	970	480	50.5%
P#14-2	2580	1290	50.0%
Total	11218	5577	50.3%

mutants are classified automatically into 2 “equivalent” and 24 “killed”. We noted from these results that using SOMs approaches [18] reduces in the number of equivalent mutants to less than half and applying DSE is able to detect approximately 100% of both of the examined first- and second-order equivalent mutants in this tested program example.

B. EXPERIMENTAL RESULTS

To demonstrate the competence of our study in classifying the mutants and detecting equivalent mutants, we compare our results with the manual results and with the previous results [21] as in section IV.B.4. The empirical results of our study illustrate that our technique can classify mutants automatically into killed (normal and naïve) and alive (equivalent and strong). In addition, it decreases the number of SOMs mutants to less than half compared with FOMs see Table 7. Besides, it identifies approximately 94% of EM that manually detected where our algorithm automatically detected 224 ESOMs from 238 ESOMs that analyzed manually. This percentage is considered a high percentage comparing with the previous results of Kintis, and Malevris [21] which can detect approximately 70% of equivalent mutants.

1) GENERATING FOMs AND SOMs

We used 14 tested programs between small-sized programs with one class and large-sized programs with many classes and lines of code. We first apply all traditional-level mutation operators of the Muclipse tool on each tested program to generate the FOMs. After generating FOMs, we apply the SCWR Algorithm [18] to generate SOMs for these FOMs. We record the number of FOMs and SOMs for each program and the reduction ratio of SOMs comparing with FOMs in Table 7. The results given in Table 7 showed that the number of SOMs is reduced by approximately 50% comparing to number of FOMs.

2) CLASSIFYING SOMs MANUALLY

Generating SOMs by SCWR Algorithm [18] reduces the number of mutants by a high percentage (app. 50%) and this

TABLE 8. SOMs classification manually.

Tested Program	#SOMs	#killed	#Equivalent
P#1	54	51	3
P#2	23	20	3
P#3	41	40	1
P#4	26	24	2
P#5	42	38	4
P#6	20	20	0
P#7	26	23	3
P#8	23	22	1
P#9	35	33	2
P#10	50	45	5
P#11	280	265	15
P#12	2942	2852	90
P#13-1	170	160	10
P#13-2	75	69	6
P#14-1	480	453	27
P#14-2	1290	1224	66
Total	5577	5339	238

TABLE 9. SOMs classification automatically.

Tested Program	SOMs	Killed		Alive	
		¹ Nr	² Nv	³ St	⁴ Eq
P#1	54	6	45	1	2
P#2	23	5	15	0	3
P#3	41	14	21	5	1
P#4	26	12	12	0	2
P#5	42	13	22	3	4
P#6	20	6	12	2	0
P#7	26	5	14	4	3
P#8	23	6	16	0	1
P#9	35	9	24	1	1
P#10	50	15	29	1	5
P#11	280	116	140	11	13
P#12	2942	1281	1548	25	88
P#13-1	170	57	94	9	10
P#13-2	75	23	39	7	6
P#14-1	480	180	265	10	25
P#14-2	1290	439	772	19	60
Total	5577	2187	3068	98	224

¹Nr: Normal; ²Nv: Naïve; ³St: Strong; ⁴Eq: Equivalent

reduction leads to a reduction in the number of the EM. In this stage, we manually classify SOMs into “killed” and “Equivalent”. The results given in Table 8 show that 238 ESOMs were manually detected of a total of 5577 SOMs.

3) CLASSIFYING SOMs AUTOMATICALLY

We classify in this stage SOMs automatically by our DEM-DSE algorithm. The empirical results are recorded in Table 9. Our algorithm classifies mutants into “alive” mutants which are classified into “equivalent” and “strong” and “killed” mutants which are classified into “normal” and “naïve” depending on the rules described in section IV. The results illustrated that our algorithm is able to classify SOMs and detects ESOMs. Our technique classifies the 5577 SOMs mutants into 224 equivalent mutants, 98 strong mutants, 2187 normal mutants, and 3068 naïve mutants (Table 9).

4) EVALUATE THE EFFECTIVENESS OF OUR DEM-DSE TECHNIQUE

In this stage, we evaluate our technique based on the manual results and the previous results [21]. Table 10 shows the

TABLE 10. The number of esoms that analyzed manually and by DEM-DSE.

Tested Program	#SOM	# SOM Manual	# ESOM Automatic	Ratio of DEM-DSE to Manual
P#1	54	3	2	66.7%
P#2	23	3	3	100.0%
P#3	41	1	1	100.0%
P#4	26	2	2	100.0%
P#5	42	4	4	100.0%
P#6	20	0	0	--
P#7	26	3	3	100.0%
P#8	23	1	1	100.0%
P#9	35	2	1	50.0%
P#10	50	5	5	100.0%
P#11	280	15	13	86.7%
P#12	2942	90	88	97.8%
P#13-1	170	10	10	100.0%
P#13-2	75	6	6	100.0%
P#14-1	480	27	25	92.6%
P#14-2	1290	66	60	90.9%
Total	5577	238	224	94.1%

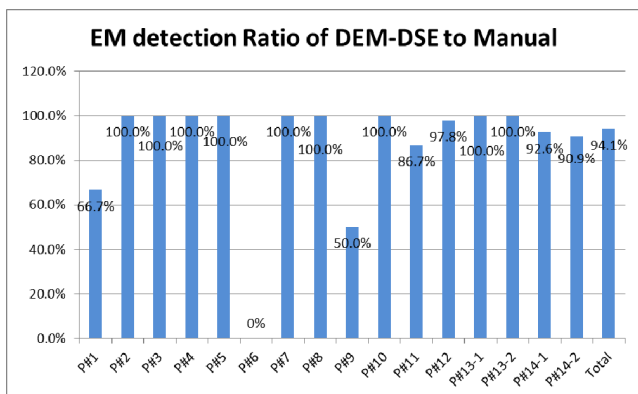


FIGURE 7. EM Detection ratio.

ratio of the detected ESOMs comparing with the manually analyzed ESOMs. The results in Table 10 and Figure 7 show that our technique is able to detect 224 ESOMs from 238 ESOMs analyzed manually. In other words, our technique is more effective than others in detecting ESOMs that manually analyzed by a percentage of 94%. This percentage is high compared with the previous techniques in [21] such that the previous approach [21] detected the ESOMs with approximately 70% of the examined equivalent mutants. In addition, the DEM-DSE technique detects 100% of equivalent mutants for 9 of the 14 subject programs.

From the above discussion, one can conclude that the EM problem can be handled as a static analysis problem or as a dynamic execution based problem. Nica and Wotawa [43] and Pan [42] used static analysis to handle this problem. While in our work, we handled this problem as a dynamic execution problem. We recommend other researchers to do more work in this area using dynamic execution because it overcomes many challenges of static analysis.

The proposed method executes the target program against randomly generated test values. Then, it collects the set

of covered constraints. By solving the complementary constraints (where, $x \geq y$ is the complementary constraint of $x < y$) of the collected ones, the method can find new test values that cover the complementary path of the covered path. By iterating this process, all branches of the target program will be covered.

C. THREATS TO VALIDITY

1) EXTERNAL VALIDITY

The key external threat to validity is applying dynamic symbolic execution to real programs because of the complication of the path constraints that are hard to be solved. This problem can be eventually overcome using powerful constraints solver. Furthermore, control flow dependence and state explosion can be external threats to validity. Therefore, the impacts of control flow dependence and state explosion on the proposed method will be studied in future work. Besides, 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 the large-sized or real programs.

2) INTERNAL VALIDITY

The key internal threats to validity is the construction of stillborn mutants. Although we manually discarded these mutants, this procedure is a time-wasting procedure and could be inaccurate.

VI. CONCLUSION

This paper introduced a new technique for detecting the equivalent mutants. This technique is a combination of second-order mutation testing and dynamic symbolic execution technique. The aim of this technique is detecting the equivalent mutants automatically instead of manual detection. This leads to a reduction in the time and effort which are consumed in manual detection. Using second-order mutation testing helped in decreasing the number of mutants to less than half and therefore the number of equivalent mutants is reduced. Also, using a dynamic symbolic execution technique helped in detecting the equivalent mutants and classifying the mutants to killed (which are classified into normal or naïve) or alive (which classified into equivalent or strong). The results proved that the proposed technique is an effective technique in detecting approximately 94% of equivalent mutants that were determined manually. This percentage is a high percentage comparing with the previous EM detecting techniques [21], [22], [26]. In addition, the DEM-DSE technique detects 100% of equivalent mutants for 9 of the 14 subject programs. In the future work, we will focus on using dynamic symbolic execution for generating test data for killing higher-order mutants. Besides, the future

work will study the impacts of control flow dependence, state explosion, stubborn mutants, and dominator mutants on the proposed method.

REFERENCES

- [1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.
- [2] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.
- [3] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Softw. Test., Verification Rel.*, vol. 19, no. 2, pp. 111–131, Jun. 2009.
- [4] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proc. 17th Asia Pacific Soft. Eng. Conf. (APSEC)*, Nov./Dec. 2010, pp. 300–309.
- [5] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the Equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 23–42, Jan. 2014.
- [6] D. Baldwin and F. G. Sayward, "Heuristics for determining Equivalence of program mutations," Yale Univ., New Haven, CT, USA, Tech Rep. 276, 1979.
- [7] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect Equivalent mutants," *Software Test., Verification Rel.*, vol. 4, no. 3, pp. 131–154, 1994.
- [8] A. J. Offutt and J. Pan, "Detecting Equivalent mutants and the feasible path problem," in *Proc. 11th Annu. Conf. Comput. Assurance (COMPASS)*, Jun. 1996, pp. 224–236.
- [9] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of Equivalent mutants," *Softw. Testing, Verification Rel.*, vol. 9, no. 4, pp. 232–262, Dec. 1999.
- [10] L. du Bousquet and M. Delaunay, "Towards mutation analysis for Lustre programs," *Electron. Notes Theor. Comput. Sci.*, vol. 203, no. 4, pp. 35–48, Jun. 2008.
- [11] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Softw. Testing, Verification Rel.*, vol. 9, no. 4, pp. 205–232, Dec. 1999.
- [12] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Mutation Testing for the New Century*, W. E. Wong, Ed. Norwell, MA, USA: Kluwer, 2001, pp. 5–13.
- [13] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the Equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation (Lecture Notes in Computer Science)*, vol. 3103. Berlin, Germany: Springer, 2004, pp. 1338–1349.
- [14] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Proc. 3rd Int. Conf. Softw. Testing, Verification, Validation Workshops*, Apr. 2010, pp. 90–99.
- [15] A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero, "Bayesian-learning based guidelines to determine Equivalent mutants," *Int. J. Soft. Eng. Knowl. Eng.*, vol. 12, no. 6, pp. 675–689, Dec. 2002.
- [16] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of Equivalent mutants," *Proc. Int. Conf. Softw. Testing, Verification, Validation Workshops*, Denver, CO, USA, Apr. 2009, pp. 192–199.
- [17] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, New York, USA, Jul. 2009, pp. 69–80.
- [18] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Reducing the cost of Higher-Order Mutation Testing," *Arabian J. Sci. Eng.*, vol. 13, no. 12, pp. 7473–7486, Dec. 2018.
- [19] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 3, pp. 215–222, Sep. 1976.
- [20] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. ESEC/FSE*, Sep. 2005, pp. 263–272.
- [21] M. Kintis and N. Malevris, "Using data flow patterns for Equivalent mutant detection," in *Proc. IEEE Int. Conf. Softw. Test., Verification, Validation Workshops*, Mar./Apr. 2014, pp. 196–205.
- [22] M. Kinitis, and N. Malevris, "Identifying more Equivalent mutants via code similarity," in *Proc. 20th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2013, pp. 180–188.
- [23] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order Equivalent mutants via second order mutation," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 701–710.
- [24] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.
- [25] A. O. Akinde, "Using higher order mutation for reducing Equivalent mutants in mutation testing," *Asian J. Comput. Sci. Inf. Technol.*, vol. 18, no. 3, pp. 13–18, 2012.
- [26] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verification Rel.*, vol. 7, no. 3, pp. 165–192, Sep. 1997.
- [27] D. Schuler and A. Zeller, "(Un-)covering equivalent mutants," in *Proc. 3rd Int. Conf. Softw. Testing, Verification Validation*, Apr. 2010, pp. 45–54.
- [28] W. J. Duran and C. S. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, Jul. 1984.
- [29] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Advances in Computer Science—ASIAN Higher-Level Decision Making (Lecture Notes in Computer Science)*, vol. 3321. Berlin, Germany: Springer, 2004, pp. 320–329.
- [30] (2008). *MuClipse Internet*. [Online]. Available: <http://muclipse.sourceforge.net>
- [31] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 13th Eur. Conf. Found. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 212–222.
- [32] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: An automated class mutation system," *Softw. Test., Verification Rel.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.
- [33] A. S. Ghiduk, "Using evolutionary algorithms for higher-order mutation testing," *Int. J. Comput. Sci.*, vol. 11, no. 2, pp. 93–104, Mar. 2014.
- [34] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *Proc. 14th Asia-Pacific Softw. Eng. Conf.*, Dec. 2007, pp. 41–48.
- [35] A. S. Ghiduk, "Automatic generation of object-oriented tests with a multistage-based genetic algorithm," *J. Comput.*, vol. 5, no. 10, pp. 1560–1569, Oct. 2010.
- [36] A. S. Ghiduk and M. R. Girgis, "Using genetic algorithms and dominance concepts for generating reduced test data," *Informatica*, vol. 34, no. 3, pp. 377–385, 2010.
- [37] M. Kintis, "Effective methods to tackle the equivalent mutant problem when testing software with mutation," Ph.D. dissertation, Dept. Inform., Athens Univ. Econ. Bus., Athina, Greece 2016.
- [38] W. Orzeszyna, "Solutions to the equivalent mutants problem: A systematic review and comparative experiment," M.S. thesis, School Comput., Blekinge Inst. Technol., Karlskrona, Sweden, 2011.
- [39] W. Q. Zhang, D. W. Gong, and X. J. Yao, "Evolutionary generation of test data for many paths coverage based on grouping," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2222–2233, Dec. 2011.
- [40] X. Yao, D. Gong, and G. Zhang, "Constrained multi-objective test data generation based on set evolution," *IET Softw.*, vol. 9, no. 4, pp. 103–108, Aug. 2015.
- [41] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 82–96, Jan. 2017.
- [42] J. Pan, "Using constraints to detect equivalent mutants," M.S. thesis, Dept. ISSE, George Mason Univ., Fairfax, VA, USA, 1994.
- [43] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," *Electron. Proc. Theor. Comput. Sci.*, vol. 86, pp. 1–8, Jul. 2012.



AHMED S. GHIDUK received the B.Sc. degree from Cairo University (Beni-Suef Branch), Egypt, in 1994, the M.Sc. degree from Minia University, Egypt, in 2001, and the Ph.D. degree from Beni-Suef University as a joint work with the College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, in 2007. He is an Associate Professor with the Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Egypt. He is currently an Associate Professor with the College of Computers and Information Technology, Taif University, Saudi Arabia. His research interests include software engineering, search-based software engineering, software testing, mutation testing, higher-order mutation testing, weak mutation testing, test data generation, requirements engineering, and genetic algorithms.



MOHEB R. GIRGIS received the B.Sc. degree from Mansoura University, Egypt, in 1974, the M.Sc. degree from Assuit University, Egypt, in 1980, and the Ph.D. degree from the University of Liverpool, England, in 1986. He is currently an Associate Professor with Minia University, Egypt, where he is also a Professor. His research interests include software engineering, information retrieval, genetic algorithms, and networks. He is a member of the IEEE Computer Society.



MARWA H. SHEHATA received the B.Sc. and M.Sc. degrees from Beni-Suef University, Egypt, in 2009 and 2014, respectively, where she is currently pursuing the Ph.D. degree. She is currently a Lecturer with the Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University. Her research interests include software engineering and higher-order mutation testing.

...