Using Evolutionary Algorithms for Higher-Order Mutation Testing

Ahmed S. Ghiduk[1, 2]

**[1] College of Computers and Information Technology, Taif University
Saudi Arabia**

**[2] Department of Mathematics and Computer Science, Faculty of Science
Beni-Suef University, Egypt**

Abstract

Most software faults are complex higher-order mutants and their fixing needs more changes than first-order mutants. First-order mutants are created by inserting a single fault in the tested program. Higher-order mutants are created by injecting two or more faults in the tested program. Mutation testing has been developed to generate test inputs to kill the mutants of the tested program. Evolutionary algorithms have been effectively used in many software testing activities especially producing the required test inputs. In this paper, we introduce a genetic algorithm based technique to aid the automatic generation of test inputs for killing higher-order mutants. The proposed technique includes two policies: the first policy aims at killing the first-order mutants, and the second policy aims at killing the higher-order mutants. In addition, we introduce two new algorithms to generate the higher-order mutants. The paper also presents the results of the experiments that have been carried out to evaluate the effectiveness of our technique with its two policies. The results of the conducted empirical study showed that our proposed technique is more efficiency than random tests generation techniques in killing higher-order mutants.

Keywords: *Mutation Testing, First-Order Mutants, Higher-Order Mutants, Test-Inputs Generation, Genetic Algorithms.*

1. Introduction

Approximately 90% of faults in software are complex faults of higher-order mutants and their fixing needs several changes [1]. The space of higher-order mutants is wider than the space of first-order mutants. Mutation testing has been developed by DeMillo et al. [2] and Hamlet [3]. It has been developed to find test inputs to kill the seeded mutants in the program under test [4]. Mutation testing motivation is that injected faults represent errors that programmers often create. Many mutation testing techniques have been developed to consider the first-order mutants (FOMs) which are created by the injection of unique fault in the tested program [5]. FOMs represent simple faults which are easily killed. Higher-order mutants (HOMs) are complex faults which are produced by inserting two or more faults in the tested program [6]. Higher-order mutation testing techniques are proposed by Jia and Harman [6] and used to study the interactions between defects and their impact on software testing for fault detecting [6].

Achieving high mutant-killing ratios is important in mutation testing and requires generating high-quality test inputs which is a crucial issue. Although manually written test inputs are valuable, they are not sufficient to prevent faults. Therefore, automatic generation of test inputs is required to complement manually written test inputs. Many automatic test-inputs generation techniques have been presented, some of which [7, 8, 9, 10] generate test inputs randomly; other techniques [11, 12, 13] use code coverage as the test criterion and generate test inputs to satisfy this test criterion; a third approach [5, 14, 15, 16, 17] use mutation testing as the test criterion and generate test inputs to kill mutants. Jia and Harman [18] provided a comprehensive analysis of trends and results of mutation testing techniques. Guiding the automatic generation of test inputs using mutant killing is intractable process [14]. Therefore, many of the existing mutation testing techniques use the concepts of weak mutation testing [5] for generating test inputs and improving the probability of mutant killing. Mutation testing techniques have been utilized to evaluate the quality of test data [15]. In addition, mutation testing requires fewer number of tests than many white box testing criteria such as edge-pair, all-uses and prime path coverage [19]. Mutation testing is more robust than code coverage in evaluating the effectiveness of test data [16]. In practice, there are many challenges that face the existing mutation-testing techniques. The automatic generation of test inputs for mutant killing is one of the main challenges for mutation testing techniques. Although there are techniques [5, 14] aiding test-inputs generation for killing mutants, they have three main drawbacks. First, these techniques construct a whole constraint system for each weak mutant killing, making it costly to generate test inputs for a large number of mutants. Second, these techniques are based on solving statically constructed constraint systems, not being able to handle programs with

complex data structures, non-linear arithmetic, or array indexing with non-constant expressions. Third, these techniques don't consider higher-order mutants killing. These limitations cause the existing mutant killing based test generation techniques to be inapplicable for real world programs, and not to be widely used in practice.

In 2009, Jia and Harman [6] proposed the concepts of higher-order mutation testing. Harman et al. [1, 20] introduced the first policy to use search-based techniques for higher-order mutation testing (HOMT). This strategy uses the genetic programming algorithms to find test data to kill higher-order mutants in C programs. The results of the experiments showed the efficacy of search-based techniques in killing higher-order mutants. Harman et al. [21] introduced mutation-based test data generation approach that combines dynamic symbolic execution and hill climbing. This technique targets strong mutation adequacy. Kapoor [22] studied the subsuming relation between the first-order mutants and the higher-order mutants. Kapoor's technique did not consider higher-order mutants killing. Akinde [23] presented an empirical study to show that higher-order mutation reduces the number of equivalent mutants.

From the above discussion, it is clear that little attention has been given in literature to search-based test data generation for killing the higher-order mutants. In addition, Langdon et al. [1] reported the problem of non-determinism in mutation testing for the first time. The problems associated with a mutant behaving differently between different runs were discussed in the context of the mutant causing a local variable not to be initialized and hence to take essentially random values on different runs. This in turn could cause the mutant to pass tests on some runs and fail the same tests (i.e., be killed) on others. The problems of non-determinism in mutation testing and the equivalent higher-order mutants have not been handled yet.

Evolutionary Algorithms have been successively used in many software testing activities showing significant robustness in producing the required test inputs. Only genetic programming and hill climbing [1] [21] have been used in higher-order mutation testing. Jia and Harman [24] reported their empirical study to identify subsuming HOMs using three search based algorithms: greedy algorithm, genetic algorithm and hill climbing algorithm. Although genetic algorithms (GAs) are considered the most powerful and the most widely employed evolutionary algorithms in search-based software testing, they have not been used in generating test inputs to kill higher-order mutants.

The main contributions of this paper are: 1) introducing a genetic algorithm based approach for generating test inputs for killing the first-order mutants and higher-order mutants; 2) using the proposed approach to implement a testing tool for automatic generation of test inputs to kill first-order mutants and higher-order mutants; 3) using the implemented tool to perform set of empirical studies to answer the following research questions:

RQ1: How effective is our proposed genetic algorithm in generating test inputs to kill non-equivalent mutants of orders one, two, and three?

RQ2: How effective is our higher-order mutation testing technique in generating test inputs compared to random testing techniques?

The rest of this paper is organized as follows. Section 2 gives some basic concepts and definitions. Section 3 describes the proposed genetic algorithm for automatic generation of test inputs for mutation testing. Section 4 describes the phases of our proposed genetic algorithm based mutation testing system. Section 5 presents the results of the experiments that are conducted to evaluate the effectiveness of the proposed GA-based technique compared to the random-based test inputs generation techniques. Section 6 discusses the related work. Section 7 gives the conclusion and future work.

2. Background

We introduce here some basic concepts that will be used throughout this work.

2.1 Mutation testing:

In mutation testing, a set of faulty programs p', called mutants, is generated by seeding faults into the original program p. A mutant is generated by making a single small change to the original program. For example, Table 1 shows a first-order mutant in the mutated program p' generated by changing the *and* (&&) operator in the original program p into the *or* (||) operator in the mutated p'. In addition, Table 1 gives a second-order mutant by changing two operators (&&) and (>) in p into (||) and (<) in p'. A transformation rule that generates a mutant from the original program is known as a mutation operator [6].

Table 1: An Example of Mutation Operation

Original Program p	*Mutated Program p'*	
	First Order Mutant	*Second Order Mutant*
if (a >0&& b> 0)	if (a>0 ‖ b > 0)	if (a > 0 ‖ b < 0)

Each mutated program p' will be executed using a test set T. If the result of running p' is different from the result of running the original program p for any test case in T (i.e., $p'(t) \neq p(t)$ for any t of T), then the mutated program p' is said to be "killed", otherwise it is said to have "survived". The adequacy level of the test set T can be measured by a mutation score [25] that is computed in terms of the number of mutants killed by T as follows.

$$MS(P,T) = \frac{\#\ of\ killed\ Mutants}{Total\ no.\ of\ Mutants - no.of\ Equivalent\ Mutants} \quad (1)$$

The aim of mutation testing is finding the test set T [2, 3].

2.2 Higher-Order Mutation Testing:

Higher-order mutation (HOM) testing is a generalization of traditional mutation testing. Higher-order mutants are constructed by inserting one or more of a given set of first order mutants into the tested program [20].

2.3 Classification of Higher-Order Mutants:

Higher-order mutants can be classified into six categories based on the way that they are 'Coupled' and 'Subsuming', as shown in Figure 1. Coupled means: complex errors are coupled to simple errors, and coupling effect hypothesis states that test input sets that detect simple types of faults are sensitive enough to detect more complex types of faults [2]. A subsuming HOMs is one in which the first-order constituent mutants partly mask one another. Therefore, a subsuming HOM is harder to kill than the first-order mutants from which it is constructed.

The area in the big circle at the centre of Figure 1 represents the domain of all HOMs. The sub-diagrams surrounding the central region illustrate each category for case of second-order mutant. In this case, there are two FOMs f_1 and f_2, and h denotes the HOM constructed from the FOMs f_1 and f_2. The two regions depicted by each sub-diagram represent the test sets containing all the test cases that kill FOMs f_1 and f_2. The shaded area represents the test set that contains all test cases that kill HOM h.

According to the coupling effect hypothesis, if a test set that kills the FOMs also includes tests that kill the HOM, then the HOM is called a 'coupled HOM', otherwise it is called a 'de-coupled HOM'. In Figure 1, a sub-diagram is a 'coupled HOM' if it contains an area where the shaded region overlaps with the un-shaded regions. For example the sub-diagrams (a), (b) and (f) are 'coupled HOMs', while diagrams (c) and (d) are 'de-coupled HOMs', because the shaded region in (c) and (d) don't overlap with the un-shaded regions. Diagram (e) is a special case of a 'de-coupled HOM', because there is no test case that can kill the HOM. The HOM in (e) is an equivalent mutant.

According to subsuming definition, the subsuming HOMs can be represented as in diagrams (a), (b) and (c) where the shaded area is smaller than the area of the union of the two un-shaded regions. By contrast, (d), (e) and (f) are non-subsuming. The subsuming HOMs can be classified into strongly subsuming HOMs and weakly subsuming HOMs. By definition, if a test case kills a strongly subsuming HOM, it guarantees that its constituent FOMs are killed as well. Therefore, if the shaded region lies only inside the intersection of the two un-shaded regions, it is a strongly subsuming HOM, as depicted in (a), otherwise, it is a weakly subsuming HOM, as depicted in (b) and (c).

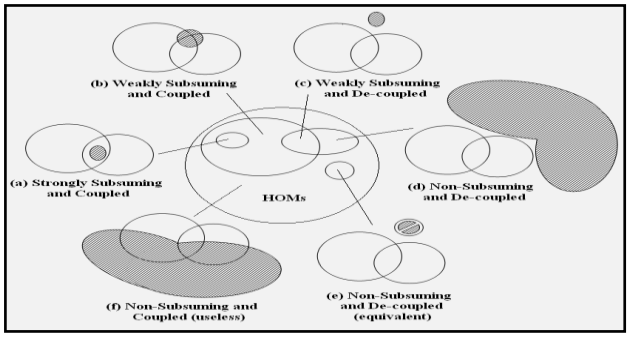

Figure 1: Higher Order Mutants Categories.

The formal definitions of the six HOMs are defined below. Let h be a HOM, constructed from FOMs $f1$, ..., fn. Assume the existence of a test set T. T is the set of all test cases under consideration. T_h is the subset of T that kills the HOM h, while T_1, ..., T_n are the subsets of T that kill the constituent FOMs f_1, ..., f_n respectively.

a. Strongly Subsuming and Coupled $T_h \subset \bigcap_i T_i \ and\ T_h \neq \emptyset$.

b. Weakly Subsuming and Coupled $|T_h| < |\bigcup_i T_i|, T_h \neq \emptyset$ $and\ T_h \cap \bigcup_i T_i \neq \emptyset$

c. Weakly Subsuming and De-Coupled

$$|T_h| < |\bigcup_i T_i|, T_h \neq \emptyset\ and\ T_h \cap \bigcup_i T_i = \emptyset.$$

d. Non-Subsuming and De-coupled

$$|T_h| \geq |\bigcup_i T_i|, T_h \neq \emptyset\ and\ T_h \cap \bigcup_i T_i = \emptyset.$$

e. Non-Subsuming and De-coupled (Equivalent) $T_h = \phi$.

f. Non-Subsuming and Coupled (Useless)

$$|T_h| \geq |\bigcup_i T_i|, and\ T_h \cap \bigcup_i T_i \neq \emptyset.$$

2.4 The Method of Mutation Testing:

The input parameters to the mutation testing are the tested program, a set of mutation operators, and a set of test inputs, T. Initially, the program under test must be executed with the test set T to show that it is correct and produces the desired outputs. If not, then program under test contains faults, which should be corrected before resuming the process.

The next stage generates set of mutants of the tested program by seeding faults in it. The seeded faults are generated by applying the mutation operators.

The generated mutants are then executed with all tests in T and their outputs compared against the outputs of the original program. If a mutant produces a different result from the tested program for any test, then the mutant is said to be "killed", otherwise it is said to have "survived".

3. A SEARCH-BASED ALGORITHM FOR HOM TESTING

This section describes the proposed search-based algorithm for automatic test inputs generation for higher-

order mutation testing. This algorithm uses the concepts of genetic algorithms to search for test inputs that kill the first-order mutants and higher-order mutants.

3.1 Chromosome Representation

The proposed genetic algorithm uses a binary vector as a chromosome to represent the inputs values of the tested program. The length, m, of this binary vector depends on the required precision and the domain length for each input variable.

Suppose we desire to generate test inputs 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 $(b_i - a_i) \times 10^{d_i}$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \times 10^{d_i} \leq 2^{m_i} - 1$. Then, a representation having each variable x_i coded as a binary vector of length m_i clearly satisfies the precision requirement. The mapping from the binary vector into a real number x_i from the range $[a_i, b_i]$ is performed by the following formula:

$$x_i = a_i + x_i' \times \frac{b_i - a_i}{2^{m_i} - 1} \qquad (2)$$

where x_i' is the decimal value of the binary vector (26).

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 formula (2):

$$x_i = a_i + int(x_i' \times \frac{b_i - a_i}{2^{m_i} - 1}) \qquad (3)$$

Now, each chromosome is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$; the first m_1 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 2 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 4 decimal places for each variable. The domain of variable x has length 15.1; the precision requirement implies that the range [-3.0, 12.1] should be divided into at least 15.1×10000 equal size ranges. This means that 18 bits are required as the first part of the chromosome: $2^{17} < 151000 \leq 2^{18}$. The domain of variable y has length 1.7; the precision requirement implies that the range [4.1, 5.8] should be divided into at least 1.7×10000 equal size ranges. This means that 15 bits are required as the second part of the chromosome: $2^{14} < 17000 \leq 2^{15}$. The total length of a chromosome 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.

Using formula (2), first 18 bits, 010001001011010000, represents x =1.0524, and 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.

3.2 Initial population

As mentioned above, each chromosome is represented by a binary vector of length m. We randomly generate *ps* of m-bit vectors to represent the initial population, where *ps* is the population size which is experimentally determined. Each chromosome is converted to k decimal numbers representing values of k input variables $x_1, \ldots, x_k$ by using formula (2) or (3).

3.3 Evaluation Function

To measure the robustness of the test inputs to kill the FOMs or HOMs, the algorithm uses a fitness function to evaluate these test inputs. We define the killing ability of a set of test inputs T_i (i = 1, …, population size) as follows:

$$\text{Killing Ability } (T_i) = \frac{\text{number of killed mutants by } T_i}{\text{total number of mutants}} \qquad (4)$$

The fitness value of a set of test inputs T_i is its killing ability. The target of the proposed genetic algorithm is maximizing this fitness function.

3.4 Selection

After computing the fitness of each test input in the current population, the algorithm uses the cumulative fitness method [26] to select test inputs from the members of the current population to be parents of the new population.

3.5 Recombination

The algorithm uses two genetic operators, crossover and mutation [26], which are the key to the power of GAs. These operators create new individuals from the selected parents to form a new population.

Crossover: It operates at the individual level with a pre-determined probability *xp*. During crossover, two parents (chromosomes) exchange some information at a random position in the chromosome to produce two new offspring. Any offspring that does not lie inside the domain of the required variables will be discarded.

Mutation: It is performed on a gene-by-gene basis. Mutation always operates after the crossover operator, and changes each gene with the pre-determined probability *mp*. Every gene (in all chromosomes in the whole population) has an equal chance to undergo mutation. A gene is mutated by changing its value from 0 to 1 and vice versa. If the mutated chromosome does not lie inside the domain of the required variables, it will be discarded.

3.6 Stop Condition

The algorithm will stop if the maximum number of generations is reached or if there isn't any enhancement in the evaluation of the generated population. In our empirical study, the algorithm stops if 97% of mutants (or more) are killed.

4.The Proposed GA-Based Higher-order Mutation Testing System (GAMTS)

This section describes the phases that form the proposed GA-based mutation testing system (*GAMTS*). The system is written in Java and consists of the following phases:

1. Program Analysis Phase.
2. Mutants Generation Phase.
3. Test Inputs Generation Phase.
4. Test Execution Phase.

These phases are described in more details below.

4.1 Program Analysis Phase:

This phase consists of one Module that performs the following tasks: 1) reading the Java source code of the program under test (PUT.java), 2) identifying the number of input variables and their data types.

Inputs of this phase: The input of this phase is the source code of the program under test in Java (PUT.java).

Output of this phase: is list of the input variables, and their data types. The outputs of this phase are passed to the Test Inputs Generation Phase. In addition, this module passes the source code of the program under test to the Mutants Generation Module.

Figure 2.(a) shows a Java example program which reads three integers and finds the maximum value. This phase finds for this example program (Maximum.java) the number of required inputs which is 3 and their data types which are int.

4.2 Mutants Generation Phase:

This phase applies the *MuJava* tool to achieve its tasks. *MuJava* automatically generates FOMs for both traditional mutation testing and class-level mutation testing for Java programs [27]. Table 2 shows the first set of mutation operators: the 22 “mothra” Fortran mutation operators [28]. *MuJava* [29] uses a subset of the 22 “mothra” Fortran mutation operators to generate the first-order mutants.

Inputs of this phase: are two files; the first is the Java source code of the program under test (PUT.java) and the second is the class file (PUT.class).

Output of this phase: is set of mutated versions of the Java source code of the program under test. Each mutated program is seeded by single fault which can be selected from an available list of Java mutation operators (transformations) [29].

Tasks of this phase are: 1) creating set of FOMs by seeding the original program with single fault. Figure 2.(b). gives an example for first-order mutant for the example program. Table 3 shows the generated FOMs by *MuJava* for the example program, Maximum.java, 2) creating set of HOMs mutants of the original program each mutated program contains higher-order mutants of order n where $n \geq 2$. Figure 3.(a) gives an example for the second-order mutant and Figure 3.(b) gives an example for third-order mutant for the example program. In fact, *MuJava* tool generates only first-order mutants. Therefore, an algorithm to generate higher-order mutants is required for our technique. Our higher-order mutants generation procedures are described below, 3) passing the mutated versions of the program under test to the test inputs generation phase.

```
1.  import java.lang.*;
2.  import java.util.Scanner;
3.  public class Maximum{
4.    public static void main(String[] args){
5.      int x, y, z;
6.      Scanner in = new Scanner(System.in);
7.       x = in.nextInt();
8.      y = in.nextInt();
9.      z = in.nextInt();
10.     z = max(x,y,z);
11.     System.out.println("max = "+z);
12.   }
13.   public static int max(int a, int b, int c){
14.     int m = a;
15.     if(b > m){
16.         m = b;
17.     }
18.     if (c > m){
19.         m = c;
20.     }
21.     return m;
22.   }
23. }
```

(a)

```
1.  import java.lang.*;
2.  import java.util.Scanner;
3.  public class Maximum{
4.    public static void main(String[] args){
5.      int x, y, z;
6.      Scanner in = new Scanner(System.in);
7.       x = in.nextInt();
8.      y = in.nextInt();
9.      z = in.nextInt();
10.     z = max(x,y,z);
11.     System.out.println("max = "+z);
12.   }
13.   public static int max(int a, int b, int c){
14.     int m = a;
15.     if(b < m){
16.         m = b;
17.     }
18.     if (c > m){
19.         m = c;
20.     }
21.     return m;
22.   }
23. }
```

(b)

Figure 2: a) Java Example Program; b) Its FOM Mutated Example.

```
1.  import java.lang.*;
2.   import java.util.Scanner;
3.   public class Maximum{
4.     public static void main(String[] args){
5.        int x, y, z;
6.        Scanner in = new Scanner(System.in);
7.         x = in.nextInt();
8.        y = in.nextInt();
9.        z = in.nextInt();
10.       z = max(x,y,z);
11.       System.out.println("max = "+z);
12.    }
13.    public static int max(int a, int b, int c){
14.       int m = a;
15.       if( b <  m ){
16.             m = b;
17.       }
18.       if (c > m){
19.             m = ++c;
20.       }
21.       return m;
22.    }
23.  }
```

(a)

```
1.  import java.lang.*;
2.   import java.util.Scanner;
3.   public class Maximum{
4.     public static void main(String[] args){
5.        int x, y, z;
6.        Scanner in = new Scanner(System.in);
7.         x = in.nextInt();
8.        y = in.nextInt();
9.        z = in.nextInt();
10.       z = max(x,y,z);
11.       System.out.println("max = "+z);
12.    }
13.    public static int max(int a, int b, int c){
14.       int m = ++a;
15.       if( b < m ){
16.             m = b;
17.       }
18.       if (c >= m){
19.             m = c;
20.       }
21.       return m;
22.    }
23.  }
```

(b)

Figure 3: a) Second-Order Mutation and b) Third-Order Mutation for the Example Program.

Higher-order mutants generation algorithm: There are many procedures that can be implemented to generate higher-order mutants for Java programs using *MuJava*. Polo et al. [30] proposed some procedures to generate second-order mutants. In the following, we introduce two algorithms to generate higher-order mutants of order greater than or equal two. To illustrate our algorithms, we use the results in Table 3 which lists the names of the 56 mutants that the *MuJava* tool generates for the example program, *Maximum.java*. The application of the operators AOIU, AOIS, ROR, COI, and LOI generated 4, 28, 14, 2, and 8 mutants, respectively. In the following, we show our mutants generation algorithms.

Circular Incremental Algorithm (*CIA*): finds the list of mutation operators which we can apply on the program under test by checking the list of operators in *MuJava* and puts these operators in circular order. *CIA* algorithm finds the operators AOIU, AOIS, ROR, COI, and LOI for the example program and puts them in the previous order. *CIA* applies *MuJava* tool to find the list of first-order mutants for the tested program. First-order mutants of the example program are shown in Table 3 . Then, the *CIA* Algorithm generates 2-order mutants in incremental manner by applying the first operator (AOIS) on the mutants files of the second operator (AOIU) and the second operator (AOIU) on the mutants files of the third operator (ROR), and so on till the last operator (LOI) is applied on the mutants files of the first operator (AOIS). Table 4 gives the second-order mutants that is generated for the example program by the algorithm *CIA*. AOIU **(**AOIS) means that applying AOIU such that AOIS was already applied.

Table 2: Set of Mutation Operators.

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 3 : The Generated FOMs by *Mujava* for the example program.

Mutation Operator (Transformation)	# of Mutants (Frequency)
AOIU: Arithmetic Operator Replacement (Replace basic unary arithmetic operators with other unary arithmetic operators).	4
AOIS: Arithmetic Operator Insertion (Insert short-cut arithmetic operators).	28
ROR: Relational Operator Replacement (Replace relational operators with other relational operators, and replace the entire predicate with true and false).	14
COI: Conditional Operator Insertion (Insert unary conditional operators).	2
LOI: : Logical Operator Insertion. (Insert unary logical operator).	8
Total # of FOMs for the example program	56

To find the third-order mutants, *CIA* applies the mutation operators in the same circle such that the first operator AOIU in row #1 in Table 4 is applied on the mutated files of operators AOIS(ROR) in row #2 (i.e., the first operator

in a row in Table 4 is applied on the mutated files of the operators in the next row). Table 5 gives the third-order mutants generated for the example program by applying the algorithm CIA.

In this manner, the Circular Incremental algorithm can generate any other order of mutants. For example, to find fourth-order mutants CIA uses Table 5 and applies the operators in the same approach (e.g., LOI is applied of the mutated files of AOIU(AOIS(ROR)) and so on).

Table 4: The Generated 2OMs by CIA for the example program.

#	Mutation Operator (Transformation)	#of Mutants (Frequency)
1	AOIU(AOIS)	98
2	AOIS (ROR)	392
3	ROR (COI)	28
4	COI (LOI)	16
5	LOI (AOIU)	32
Total # of 2OMs		566

Table 5: The Generated 3OMs by CIA for the example program.

#	Mutation Operator (Transformation)	# of Mutants (Frequency)
1	LOI (AOIU(AOIS))	784
2	AOIU (AOIS (ROR))	1372
3	AOIS (ROR (COI))	784
4	ROR (COI (LOI))	224
5	COI(LOI (AOIU))	64
Total # of 3OMs for the example program		3164

Random N Algorithm (RNA): This algorithm randomly selects *n* different operators. Then, it applies the first operator on the tested program using *MuJava*, the second is applied on the mutants files of the first operator, and the third is applied on the results of the second and so on. To generate fourth-order mutants, *RNA* algorithm randomly selects ROR, LOI, AOIS, and COI. Then, *RNA* algorithm applies the selected operators in random order (e.g., ROR, COI, LOI, AOIS). *RNA* algorithm generates 4032 mutants of fourth order for the selected operators. If the number of all available operators is less than the required order, the *RNA* algorithm can select any operators more than one time. In the case of selecting an operator more than one time, the algorithm removes the repeated mutants.

4.3 Test Inputs Generation Phase.

This phase utilizes the proposed GA algorithm which is described in Section 3 to generate set of test inputs for killing the mutants of the program under test.

Inputs of this phase: are the program under test and its mutated versions, number of the required test inputs (input variables of the tested program), the input domains of these variables and their data types, population size *ps*, maximum number of generations *maxgen*, probability of crossover *xp*, and probability of mutation *mp*.

Outputs of this phase: include a set of test inputs that kills first-order mutants and higher-order mutants, a list of the generated test inputs, the list of killed mutants, and the list of survived mutants, if any.

Tasks of this phase: are computing the chromosome length according to the input domains of the input variables or based on its data types if the input domains are not available, and applying the proposed genetic algorithm to generate set of test inputs for mutation testing.

Test-Inputs Generation Procedure: We will use the example program which is shown in Figure 1.(a) to illustrate the test inputs generation phase. The example program needs three integers as inputs. Suppose that the three input variables are x1, x2, and x3 with input domains [1, 10], [2, 15], and [3, 20], respectively. In the following we show the steps of the proposed genetic algorithm to generate test inputs to kill the third-order mutant which is shown in Figure 3.(b).

1. The proposed GA finds the length of the chromosome (as in Section 3.1) which will be 13 bits whereas x1 needs 4 bits, x2 needs 4 bits, and x3 needs 5 bits.

2. The proposed GA generates *ps* of bit vectors to represent the initial population, where *ps* is the population size. If *ps* = 3 the initial population will be as follows.

c1 = 0101110001000, c2 = 0100011101101, c3 = 1000001100110

3. The proposed GA uses the fitness function which is proposed in Section 3.3 to evaluate the initial population as follows. The algorithm converts c1, c2, and c3 from binary form to integer values using formula (3). Table 6 shows values of the variables.

Table 6: Phenotype of the Initial Population and Its Evaluation.

↓Chromosome \ input variable→	***x1***	***x2***	***x3***	***Fitness Value***
c1	5	12	8	0.5
c2	4	7	13	0.6
c3	8	3	6	0.5
Relative Fitness		***Cumulative Fitness***	***R***	***Parents***
0.3125		0.3125	0.5123	c2 = 0100011101101
0.375		0.6875	0.7432	c3 = 1000001100110
0.3125		1	0.6123	c2 = 0100011101101

The proposed system (*GAMTS*) calls the Test Execution Phase to execute the original program and all mutants using each set of test inputs and find the ratio of killed mutants. Assume this experiment aims at killing 100 mutants of the third-order mutants which is shown in Table 5 (20 mutants from each row). Table 6 shows the fitness values of each set of inputs. It is clear that the second set of inputs is the best one in this population.

4. The algorithm calculates the cumulative fitness for each chromosome as follows.

a. Calculate the total fitness of the population:

tFitness = 0.5 + 0.6 + 0.5 = 1.6 .

b. Find the relative fitness for each individual:

rFitness(c1) = fitness(c1)/tFitness= 0.3125,

rFitness(c2) = 0.375, and rFitness(c3) = 0.3125.

c. Calculate the cumulative fitness:

cFitness(c1) = rFitness(c1) = 0.3125,
cFitness(c2)=cFitness(c1)+rFitness(c2)=0.3125+0.375=0.6875,
cFitness(c3) =cFitness(c2) + rFitness(c3) =0.6875+ 0.3125 = 1.

d. Find random number r for each individual (Table 6).

e. According to the cumulative fitness method [26] the parents of the next population will be

c2 = 0100011101101, c3 = 1000001100110, c2 = 0100011101101.

5. The algorithm applies the crossover and mutation genetic operators to find the new population as follows.

a. Find random number *r1* for each individual (Table7).

b. According to the genetic crossover method [26], if the crossover probability is 0.85, the first and the third parents will be used in the crossover process.

c. According the genetic mutation method [26], find random number r2 for each individual (Table7). Let genetic mutation probability (mp) is 0.15. If r2 < mp, the algorithm changes a random bit.

6. *GAMTS* repeats the above procedure till one of the stop conditions is satisfied.

Table7 : Crossover and Mutation of the Selected Parents.

Parents	r1	New chromosomes	r2	New chromosomes	x1	x2	x3	Fitness Value
p1= 0100011101101	0.6734	0100011100110	0.0134	0110011100110	6	7	6	0.6
p2= 1000001100110	0.3456	1000001101101	0.5456	1000001101101	8	3	13	0.4
p3= 0100011101101	0.7343	0100011101101	0.0343	0100011101111	4	7	15	0.4

4.4 Test Execution Phase.

Inputs of this phase: are the original tested program, the mutated versions of it, and the generated test inputs.

Tasks of this phase: include executing the tested program and its mutants using the generated test inputs, recording the killed mutants, and checking the coverage of all mutants.

Outputs of this phase: are the number of killed mutants, total number of mutants, and passing the outputs to the fitness calculating module.

5. EMPIRICAL STUDY

This section describes the empirical setup and the study we performed to evaluate our system.

5.1 Empirical Setup

Prototype: Figure 4 gives the architecture of the prototype of *GAMTS*, which consists of four modules: analysis module, mutants generation module (*MuJava*), test inputs generation module, and test execution module. All modules are written in Java language. This prototype is based on the proposed genetic algorithm which is shown in Section 3 and the mutation testing system which is presented in Section 4.

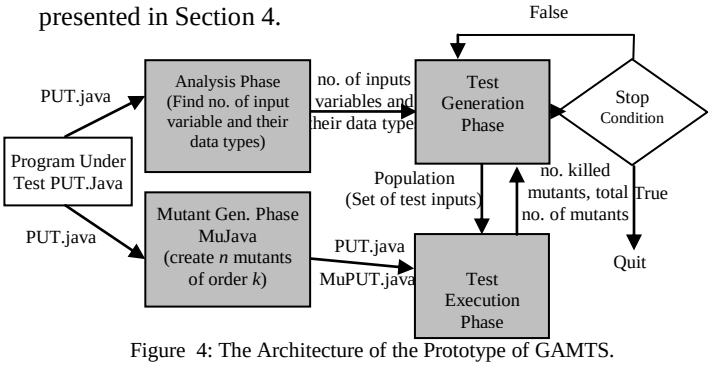

Figure 4: The Architecture of the Prototype of GAMTS.

Subject Programs: We used two sets of Java programs for our empirical studies. The first set of subject programs contains some common programs which are often used as subject programs in many software testing studies. This set of programs are Triangle, DateRange, CalDay, Select, Mid, BubSort, Power, and Remainder. The second set of subject programs are selected from the Software-artifact Infrastructure Repository (SIR) [31].

Table 8 shows details of the subject programs: The first column, Subject Program, gives a designated title of the program under test; the second column, Reference, shows some of the previous studies which used this set of subject programs; the third column, Scale, shows the number of lines of code, classes, and methods in the subject program; the fourth column, No. of FOMs, provides the number of first-order mutants in each subject program, the fifth column, Mutation Operators, gives the details of the first-order mutants. Table 9 shows the first, second, and third order mutants for each subject program.

Procedure: the empirical study is conducted as follows.

1. We run the analysis module of our prototype to find the set of input variables of the program and their data types.

2. We run *Mujava* tool to generate FOMs of the program under test. Then, we apply the *CIA* algorithm to generate the mutants of orders two and three.

3. The GA is adapted such that *maxgen = 100*, *ps =10*, *xp = 0.80* and *mp =0.15*. For fairness the random generation technique is adapted to find 10 set of test inputs 100 times.

4. The test-inputs generation module is executed, which executes *GAMTS*, to find the test inputs.

5. We use the test execution phase to execute the original tested program and its mutants using the generated test inputs and count the number of killed mutants.

5.2 Study Objectives:

We applied the study procedure to measure the efficiency of our proposed technique in killing the non-equivalent

Table 8: Subject Programs.

#	Subject Program	Reference	Scale (LOC, Classes, Methods	No. of FOMs	Mutation Operators														
					AORB	AORS	AOIU	AOIS	AODU	AODS	ROR	COR	COD	COI	SOR	LOR	LOI	LOD	ASRS
1	Select	(32), (33)	147 LOC 1C, 3 M	60	4	8	215	0	0	270	40	0	30	0	0	50	0	0	60
2	Triangle	(1), (32), (30), (34)	50 LOC 1 C, 2 M	359	36	0	3	128	0	0	119	14	0	24	0	0	35	0	0
3	Remainder	(34), (35), (36)	43 LOC 1 C, 2 M	183	20	0	10	90	0	0	31	2	0	7	0	0	23	0	0
4	CalDay	(32), (33)	30 LOC 1 C, 2 M	219	88	0	13	82	0	0	14	0	0	3	0	0	19	0	0
5	Mid	(30), (34), (36)	42 LOC 1 C, 2 M	103	0	0	6	62	0	0	14	0	0	5	0	0	16	0	0
6	Power	(34), (35), (36)	32 LOC 1 C, 2 M	79	16	0	8	40	1	0	5	0	0	3	0	0	6	0	0
7	Maximum	(34), (35), (36)	23 LOC 1 C, 2 M	56	0	0	4	28	0	0	14	0	0	2	0	0	8	0	0
8	Sort	(34), (35), (36)	33 LOC 1 C, 2 M	178	40	4	6	78	0	0	20	0	0	6	0	0	24	0	0
9	Array-Partition	SIR (31)	13 LOC 1 C, 2 M	222	28	0	11	116	0	0	14	8	0	13	0	0	32	0	0
10	Disjoint-Set	SIR (31)	35 LOC 1 C, 4 M	78	0	2	5	42	1	0	7	0	0	4	0	0	17	0	0
11	Binary-Heap	SIR (31)	72 LOC 2 C, 2 M	54	0	26	150	0	0	43	23	0	34	0	0	78	0	0	54
12	Binary-Search-Tree	SIR (31)	130 LOC 4 C, 4 M	76	0	31	250	0	0	51	30	0	28	0	0	83	0	0	76
13	Doubly-Linked-List	SIR (31)	277 LOC 1 C, 9 M	105	0	59	287	0	0	65	45	0	47	0	0	97	0	0	105

first, second, and third order mutants. This study addresses the following research questions:

RQ1: How effective is our proposed genetic algorithm in generating test inputs to kill non-equivalent mutants of orders one, two, and three?

RQ2: How effective is our higher order mutation testing technique in generating test inputs and reducing size of test suite, compared to random generation technique (RGT)?

5.3 Results and Discussion

Figure 5 shows the ratio of killed mutants of the first-order mutants, second-order mutants, and third-order mutants. The result shows that our proposed technique killed 81.8% of the first-order mutants, 90% of the second-order mutant, and 93% of the third-order mutants of the total number of mutants in all subject programs. While the random technique killed 68.1% of the first-order mutants, 77.4% of the second-order mutant, and 80.6% of the third-order mutants of the total number of mutants in all subject programs. The results of the study show the effectiveness of our proposed technique in generating test inputs to kill non-equivalent mutants of orders one, two, and three. In addition, Figure 5 shows the ratio of killed higher order mutants (2OMs, and 3OMs) by our proposed technique and the random technique. The results show that our technique outperforms the random techniques in killing the mutants of the first, second, and third order.

Figure 6 shows the ratio of killing FOMs using *GAMTS* and RGT. The results of the study show that our proposed system *GAMTS* killed higher number of FOMs than random generation technique for all subject programs.

Table 9: First, Second, and Third Order Mutants.

#	Subject Program	FOMS	2OMs	3OMs
1	Select	677	76750	3387600
2	Triangle	359	19826	358532
3	Remainder	183	4587	55456
4	CalDay	219	5129	117990
5	Mid	103	1486	11148
6	Power	79	622	6033
7	Maximum	56	566	3164
8	Sort	178	3436	24432
9	Array-Partition	222	4736	83016
10	Disjoint-Set	78	399	1564
11	Binary-Heap	408	8014	114036
12	Binary-Search-Tree	549	11502	118524
13	Doubly-Linked-List	705	24547	296335
	Total	3816	161600	4577830

5.4 Threats to Validity

There are two main external threats to validity, which are conditions that limit the ability to generalize the results of our empirical study to a larger population of subjects

programs. First, although the set of the subject programs contains some programs which have been used in many previous studies but we cannot claim that these subjects represent a random selection over the population of programs as a whole. Second, *MuJava* tool cannot generate the higher-order mutants, therefore we proposed two algorithms (*CIA* and *RAN*) based on *MuJava* to find the higher-order mutants. Therefore, the generated mutants sometimes have redundant mutants.

There are three main internal threats to validity, which are influences that can affect the dependent variables. First, we compare our technique with the random test-inputs generation technique, which may not be sufficient to evaluate the reliability of our technique. Second, the generated number of higher-order mutants is very big (161k 2OMs, $4k^2$ 3OMs). Therefore, we selected subset of this number of higher-order mutants to carry out our empirical study. Although this set of mutants is selected in a significant way, another empirical studies are required to cover all the generated higher-order mutants. Third, we consider a higher-order mutant a killed mutant if any one of its first-order mutants is killed. Although this method gives a significant ratio of discovering the higher-order mutants, it cannot discover all its first-order mutants which compose these higher-order of mutants.

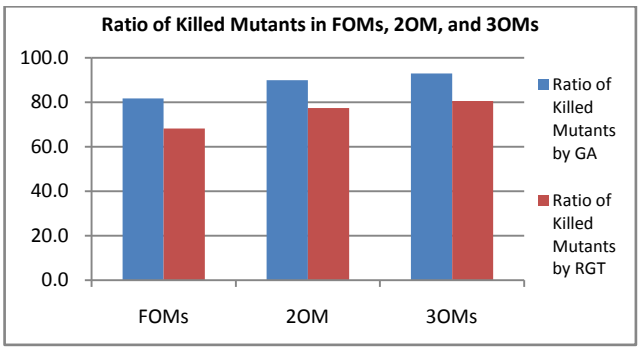

Figure 5: Ratio of Killed Mutants in FOMs, 2OM, and 3OMs.

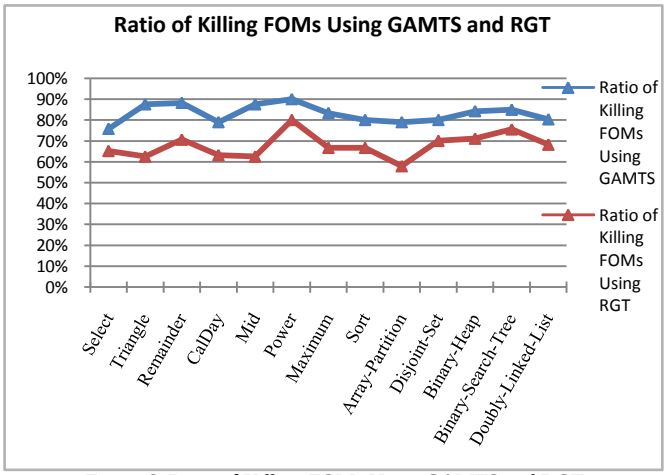

Figure 6: Ratio of Killing FOMs Using GAMTS and RGT.

6. RELATED WORK

Search-based techniques [32] have been used to generate test inputs for killing the first-order mutants. These techniques used the *mutation score* formula (1) as fitness function to evaluate the generated test inputs. These techniques didn't handle the problem of higher-order mutation testing.

Recently, a very few number of work have been done in the field of using search-based techniques for higher-order mutation testing. Langdon et al. [1] used genetic programming to find test inputs for killing the higher-order mutants in the program under test. They used two fitness functions: semantic difference and syntactic difference to evolve mutant programs. The syntactic distance sums the number of changes weighted by the actual difference. Syntactic distance places the six C comparison operations <, <=, ==, !=, >=, > in order. The distance of one comparison from another is their distance in this order plus six if they differ at all. The total distance of a mutant is the sum of the individual distances for each comparison it contains. This fitness can't find the distance for the change in descending order. Semantic distance is measured as the number of test cases for which a mutant and original program behave differently. This fitness can't find the fitness value of a single test case which is required by the search-based techniques.

Harman et al. [20, 6] defined a fitness function to capture a HOM's reduced Fragility. Let T be a set of test cases, $\{M_1, ..., M_n\}$ be a set of mutants, and the $kill(\{M_1, ..., M_n\})$ function returns the set of test cases which kill mutants $M_1, ..., M_n$. The fragility of a mutant is defined as follows: $Fragility(\{M_1, \dots, M_n\}) = \frac{|\cup_{i=1}^{n} kill(M_i)|}{|T|}$. The value of fragility lies between 0 and 1. When it equals 0 this means that there is no test case that can kill this mutant, which indicates that this mutant is potentially an equivalent mutant. The fitness of a HOM is defined as the ratio of the fragility of its HOM to the fragility of the constituent FOMs.

Jia and Harman [24] measured the fitness of the HOM using the killability of both FOMs and HOMs. Killability $= \frac{\#\text{ of test cases that kill the mutant}}{\text{total }\#\text{ of test cases}}$. The fitness of a HOM describes the ratio of the killability of the HOM to the union of the killability of each constituent FOM. The fitness function evaluates the HOM mutants rather than evaluating the test cases which is required in generating test inputs by search-based techniques. It is clear that the fitness functions of the related work are not appropriate to evaluate the test inputs generated by genetic algorithm.

7. Conclusion and Future work

In this paper, we introduced a genetic algorithm based technique to aid the automatic generation of test inputs for killing higher-order mutants. The proposed technique allows the user to choose one of two policies: the first policy aims at killing the first-order mutants, and the second policy aims at killing the higher-order mutants. In addition, we introduced two algorithms to generate the higher-order mutants. The paper presented the results of the experiments that have been carried out to evaluate the effectiveness of the system with its two policies. The results of the experiments showed that our proposed technique is effective in generating test-inputs to kill the higher-order mutants. Also, the results showed that our proposed technique is more effective than the test inputs random generation techniques. In future work we plan to perform these studies with real and large subject programs. Also, we will compare our technique with another test-inputs generation techniques rather than random technique. In addition, our future work will concern on solving the problem of explosion of higher order mutants.

References

[1] W. B. Langdon, M. Harman, Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software. 2010, Vol. 83, pp. 2416–2430.

[2] R. A. DeMillo, R. J. Lipton, F. G. Sayward. Hints on test data selection:Help for the practicing programmer. IEEE Computer. 1978, Vol. 11, 4, pp. 34–41.

[3] R. G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering SE-3. 1977, Vol. 4, pp. 279–290.

[4] K. Ayari, S. Bouktif, G. Antoniol. Automatic mutation test input data generation via ant colony. Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07, ACM. 2007, pp. 1074–1081.

[5] W. Howden. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering. 1982, Vol. 8, 4, pp. 371–379.

[6] Y. Jia, M. Harman. Higher Order Mutation Testing. Journal of Information and Software Technology. 2009, Vol. 51, 10, pp. 1379–1393.

[7] T. Chen, R. Merkel, P. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. Proceedings of Fourth International Conference on Quality Software (QSIC '04). 2004, pp. 79-86.

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. Proceedings of the 30th international conference on Software engineering (ICSE '08). 2008, pp. 71-80.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. Feedback-directed random test generation. Proceedings of the 29th international conference on Software Engineering (ICSE '07). 2007, pp. 75-84.

[10] P. Godefroid, N. Klarlund, K. Sen. DART: Directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). 2005, pp. 213-223.

[11] A. S. Ghiduk, M. R. Girgis. Using Genetic Algorithms and Dominance Concepts for Generating Reduced Test Data. Informatica Journal. 2010, Vol. 34, 3, pp. 377-385.

[12] A. S. Ghiduk. Automatic Generation of Object-Oriented Tests with a Multistage-Based Genetic Algorithm. Journal of Computers. 2010, Vol. 5, 10, pp. 1560-1569.

[13] R. DeMillo, A. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering. 1991, Vol. 17, 9, pp. 900–910.

[14] M. Liu, Y. Gao, J. Shan, J. Liu, L. Zhang, J. Sunin. An approach to test data generation for killing multiple mutants. 22nd IEEE International Conference on Software Maintenance (ICSM '06). 2006, pp. 113–122.

[15] J. Andrews, L. Briand, Y. Labiche. Is mutation an appropriate tool for testing experiments? Proceedings of 27th International Conference on Software Engineering (ICSE '05). 2005, pp. 402–411.

[16] P. Frankl, S. Weiss, C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness," The Journal of Systems & Software. 1997, Vol. 38, no. 3, pp. 235–253.

[17] M. Papadakis, N. Malevris, M. Kallia. Towards automating the generation of mutation tests. Proceedings of the 5th Workshop on Automation of Software Test (AST '10). 2010, pp. 111–118.

[18] Y. Jia, M. Harman. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering. 2011, Vol. 37, no. 5, pp. 649-678.

[19] N. Li, U. Praphamontripong, J. Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair,All-uses and Prime Path Coverage. IEEE International Conference on Software Testing Verification and Validation Workshops. 2009, pp. 220-229.

[20] M. Harman, Y. Jia, W. B. Langdon. A Manifesto for Higher Order Mutation Testing. Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW). 2010, pp. 80–89.

[21] M. Harman, Y. Jia , W. B. Langdon. Strong higher order mutation-based test data generation. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). 2011, pp. 212-222.

[22] S. Kapoor. Test case effectiveness of higher order mutation testing. International Journal of Computer Technology and Applications. 2011, Vol. 2, no.5, pp. 1206-1211.

[23] A. O. Akinde. Using higher order mutation for reducing equivalent mutants in mutation testing. Asian Journal of Computer Science and Information Technology. 2012, Vol. 2, no. 3, pp. 13–18.

[24] Y. Jia, M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. Technical Report TR-08-03,

http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-08-03.pdf. Last visit 2/2014.

[25] I. Burnstein. Practical Software Testing: A Process-Oriented Approach. Springer. 2003. ISBN-13: 978-0387951317.

[26] Z. Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs, 3rd. Springer. 1998.

[27] Y. Ma, Jeff Offutt, Y. Kwon. MuJava : An Automated Class Mutation System. 2005, Vol. 15, no. 2, pp. 97-133.

[28] K. N. King, A. J. Offutt. A Fortran Language System for Mutation-Based Software Testing. Software: Practice and Experience. 1991, Vol. 21, no. 7, pp. 685–718.

[29] Y. Ma, J. Offutt. Description of Method-level Mutation Operators for Java.

[30] M. Polo, M. Piattini, I. Rodriguez. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification and Reliabilty. 2009, Vol. 19, pp. 111–131.

[31] H. Do, S. G. Elbaum, G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. An International Journal Empirical Software Engineering. 2005, Vol. 10, no. 4, pp. 405–435. http://sir.unl.edu/portal/index.php.

[32] P. May, J. Timmis, and K. Mander. Immune and Evolutionary Approaches to Software Mutation Testing. LNCS 4628. 2007, pp. 336–347.

[33] P. May. Test Data Generation: Two Evolutionary Approaches to Mutation Testing. PhD thesis, The University of Kent at Canterbury. 2007.

[34] A. S. Ghiduk, M. J. Harrold, M. R. Girgis. Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage. 14th Asia-Pacific Software Engineering Conference. 2007.

[35] R. P. Pargas, M. J. Harrold, R. R. Peck. Test data generation using genetic algorithms. Journal of Software Testing, Verifications, and Reliability. 1999, Vol. 9, pp. 263-282.

[36] C. C. Michael, G. E. McGraw, M. A. Schatz. Generating software test data by evolution. IEEE Transactions on Software. 2001, Vol. 27, no.12, pp. 1085-1110.

Ahmed S. Ghiduk is an assistant professor at Beni-Suef University, Egypt. He received the BSc degree from Cairo University, Egypt, in 1994, the MSc degree from Minia University, Egypt, in 2001, and a Ph.D. from Beni-Suef University, Egypt in joint with College of Computing, Georgia Institute of Technology, USA, in 2007. His research interests include software engineering especially search-based software testing, genetic algorithms, and ant colony and web application testing. Currently, Ahmed S. Ghiduk is an assistant professor at College of Computers and Information Technology, Taif University, Saudi Arabia.