Improved Genetic Algorithm to Reduce Mutation Testing Cost

Muhammad Bilal Bashir and Aamer Nadeem, *Center for Software Dependability, CUST*[1]

***Abstract*— Mutation testing is a fault based testing technique that helps generating effective test cases. Mutation testing is computationally expensive because it requires executing hundreds and even thousands of mutants. In this situation search-based approaches like genetic algorithm can help to automate test case generation to reduce the cost. In this paper we present an improved genetic algorithm that can reduce computational cost of mutation testing. First we present a novel state-based and control-oriented fitness function that efficiently uses object-oriented program features to evaluate a test case. We then empirically evaluate it using our implemented tool, eMuJava, and compare it with standard fitness function. Results show that although our proposed fitness function provides detailed information about fitness of a test case but standard genetic algorithm is incapable of using that effectively to repair the test cases. So we propose new two-way crossover and adaptable mutation methods that intelligently use the fitness information to generate fitter offspring. Finally we compare the improved genetic algorithm with random testing, standard genetic algorithm, and EvoSuite. Experiment results prove that our proposed approach can find the optimal test cases in less number of attempts (reduces computational cost). Besides that it can detect software bugs from suspiciously equivalent mutants and these mutants eventually get killed (increases mutation score).**

***Index Terms*— control flow of program, genetic algorithms, mutation testing, object oriented programming, object's state, suspicious mutant, two-way crossover**

I. INTRODUCTION

Object-oriented mutation testing is a fault based testing technique that helps generating effective test case set. This has been an area of interest for the researchers and a lot of research [5, 6, 9, 10, 12, 25, and 37] has been done over the last two decades. Mutation testing is computationally expensive because it requires execution of hundreds and even thousands of variants (mutants) of the program under test. Evolutionary testing aims to automate test case generation process in order to reduce the time taken by manual test case generation. Genetic algorithm [1] is one of the approaches used in evolutionary testing that uses genetic evolution to generate solutions (test cases) to meet a certain goal (test coverage criteria). Genetic algorithm attempts to search the optimal solution from the input domain and the search is guided by its core part, the fitness function. In literature, we find several variations of fitness function [13, 14, 21, 22, 23, 24, 27, and 28] designed by researchers for genetic algorithm.

Evolutionary mutation testing combines evolutionary testing techniques (like genetic algorithm) with mutation testing to automate test case generation process and to reduce computational overhead. Evolutionary mutation testing is relatively new domain of research and it is still evolving. Researchers are still combining various approaches and methods of evolutionary and mutation testing techniques to devise best possible combination to take full advantage of both the base techniques. Our literature survey shows that existing approaches [7, 33, 34, 35, 36 and 39] do not consider some important aspects of object-oriented programs. We briefly discuss these aspects in next paragraphs.

Testing an object-oriented program is more challenging than a program written in structured programming language. An object-oriented program is usually based on one or more classes and in order to test a class, its object is created. An object comprises of two parts; data (states) and behavior (operations) and normally we are interested in testing all of its behaviors. In order to test an object's behavior, it may be required for the object to be in a certain state otherwise, it may not be possible to test that. For example, we can consider an object of stack class. If we want to test "pop" operation on the stack object, the stack must be in 'non-empty' state, which means the stack should contain at least one element in it. For the stack object to be in 'non-empty' state the 'push' operation must be invoked before the 'pop' operation and similarly in general we may require invoking multiple operations on object to gain the desired state. Considering the importance of object's state, in our earlier work [28], we propose a fitness function for genetic algorithm that takes object's state into account and uses it to evaluate a solution (test case). With the inclusion of object's state as part of the fitness, the experiments prove that the search process gets better guidance and converges to the target quickly in lesser iterations of genetic algorithm.

Control flow of a program is another important aspect, which helps determining the behavior of a program on a certain input. The control-flow information therefore can be used to check if the program exhibits the desired behavior and has followed the expected path during execution. The difference in expected and actual execution paths may indicate the presence of a logical bug (a logical mistake by the programmer, which may change the output of program) in the program. We propose [37] using control-flow information in mutation testing besides using program's output (which is based on data states). If original and mutated (modified) programs exercise different execution paths but produce same

Date Submitted: February, 2017
[1]CUST: Capital University of Science & Technology, Islamabad, Pakistan.

output, we may end up finding a potential bug in the program, which is not allowing the modified program to produce a different output on a different execution path.

We have designed a new fitness function for evolutionary mutation testing of object-oriented programs [38]. We have introduced two new values in a test case fitness including object's state and control flow information. Both of these values are calculated from execution traces of program under test and are represented as separate costs in fitness. By using object's state fitness the search process can decide if a given test case needs modification in method call sequence (responsible to allow an object to gain the desired state) before calling the method under test. After execution of mutated statement, the mutated program usually exercises a different execution path (if mutated statement causes some change) as compare to the original one. The fitness value that we calculate using control flow information tells if the mutated program is taking a different route till the end or not. The different execution path has tendency to generate different output but situation can become interesting if mutated program produces same output even though it exercises a different path. Such a situation can indicate presence of some logical error in the program. We also did an implementation of our proposal and performed brief experiments with it. We present more details about this fitness function in section IV.

In this research we extend our proposed fitness function in various dimensions. Our major contributions include the following;

- We have implemented our proposed fitness function and named the implementation as eMuJava. Besides that eMuJava provides other test case generation techniques including random testing and standard genetic algorithm for mutation based testing. We have performed experiments using eMuJava and have compared the results that we have obtained from random testing, genetic algorithm with standard fitness function, and genetic algorithm with proposed fitness function,
- We have proposed a new crossover method for genetic algorithm to get full advantage of object's state fitness. The new crossover method covers the input domain comprehensively to generate fitter offsprings for next iteration.
- We have proposed a novel adaptable mutation method for genetic algorithm. The new method adapts to the situation by changing mutation rate dynamically at runtime. We have implemented two-way crossover and adaptable mutation methods in eMuJava v.2.
- We have performed experiments using eMuJava v.2 to validate improvements in genetic algorithm. eMuJava v.2 supports four methods for mutation based test case generation. Experiments have proven that our improved genetic algorithm achieves targets in lesser iterations and can raise mutation score by detecting bugs in the program. We have also compared our approach with EvoSuite [35].

The coming sections contain more details about our contributions and other important aspects, which are necessary to explain.

The rest of this paper is organized as follows; section II provides introduction of evolutionary mutation testing and concepts of this area. Section III sheds some light on the related work and highlights their limitations and issues. Section IV presents details about our earlier work and section V presents eMuJava tool that we have implemented as a proof of concept. In section VI, we include results and analysis of our initial experiments. Section VII presents novel methods including two-way crossover and adaptable mutation for genetic algorithm. In section VIII we provide details of experiments that we have performed using extended eMuJava and compare experiment results with the competitors of eMuJava. We conclude our discussion in section IX with some future directions.

II. Evolutionary Mutation Testing

Evolutionary mutation testing is a step towards automating test case generation process in mutation testing with the help of evolutionary testing techniques like genetic algorithm. By applying evolutionary testing techniques the amount of time required by mutation testing can be significantly reduced. Mutation testing is different in nature as compared to other white-box testing techniques and it also uses a different coverage criteria. The goal in mutation testing is to achieve high mutation score by killing all the non-equivalent mutants. So we need special kind of fitness functions to complement the mutation testing and to provide better guidance to search process. Some research [7, 34, 36, and 38] has been done in this area to exploit full advantages of evolutionary testing and to overcome limitations of mutation testing techniques like computational overhead. The figure 1 shows the inputs, output and activities involved in the process.

Initially, the process takes two inputs; program under test and mutation operators to generate set of mutant programs. Each mutant contains exactly one fault (syntactic change) and the set of mutants become the targets for which Genetic algorithm generates test cases to kill them. Genetic algorithm also takes two inputs, set of mutants and initial population (set of test cases) to start its operation. Then genetic algorithm executes test cases against mutant programs and records execution traces. Then it evaluates traces to check if test cases have killed all the mutants. If some mutants survive, the genetic algorithm tries to improve the test cases using crossover and biological mutation. Then with improved set of test cases, genetic algorithm executes next iteration and this process continues until it kills all the mutants or number of maximum iterations exhaust.

Evolutionary mutation testing requires a special kind of fitness function in order to evaluate test cases it generates to kill the mutants. We find three conditions in the literature that must be satisfied in order to kill a mutant, which includes; reachability condition [7], necessity condition [2], and sufficiency condition [2]. Reachability condition requires that the test case must execute the statement containing the mutation. It is important for the test case to execute this statement; otherwise the behavior of original and mutated program remains the same. Secondly, the mutated statement must introduce some infection (necessity condition). That means, it must cause some change in behavior of the program

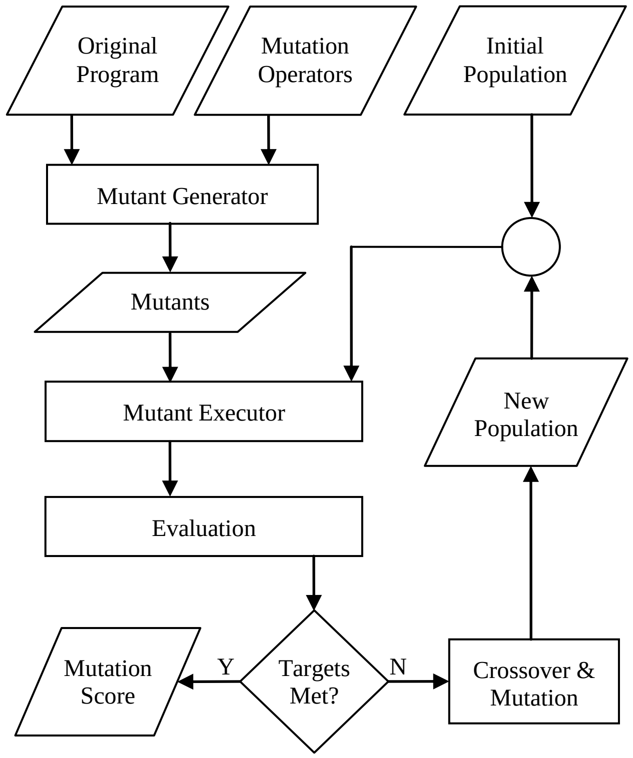


Fig. 1. Evolutionary mutation testing process flow

because this is the purpose of introducing the mutation in the original program. Finally the mutated program must produce different output after executing the mutated statement and introducing a change in behavior. If the mutated program does so, that means sufficiency condition holds.

III. Related Work

In this section we present the related research work that has been done in the area of evolutionary mutation testing.

Bottaci [7] proposes a fitness function for evolutionary mutation testing that uses three costs for three conditions, which the testing process tries to satisfy to kill a mutant. The fitness function assigns costs to a test case on the basis of its ability to satisfy these conditions. It computes reachability cost by finding the difference between goal and executed path, and by calculating the branch distance on the failed predicate. The fitness function assigns necessity cost the same way as of branch distance for reachability cost. Finally, it calculates the sufficiency cost by counting the number of same data states after the mutated statement.

Mishra, Tiwari, Kumar, and Misra [33] propose a new approach for mutation testing using elitist genetic algorithm. They extend the work of Bybro [11] and Masud [20] to generate test case for Java based programs. This approach supports unit (class) level testing and the generated test cases are in JUnit [40] format. Their approach uses test cases, which have killed some mutants, as initial population because these are good test cases and can improve performance of test case generation to a great deal.

Fraser and Zeller [34] design and introduce a novel fitness function that is purely object-oriented. They use three different costs to evaluate a test case, which misses the target and fails to kill a mutant. The first cost is distance to calling function that assigns cost to a test case that does not contain call to the function that defines mutated statement. Fraser and Zeller use a test case structure which seems different from standard object-oriented test case [14]. The second cost is distance to mutation that is calculated using approach level and branch distance, which is similar to the reachability cost we find in the work of Bottaci [7]. The third cost is mutation impact that considers unique number of methods in mutated program for which the coverage changes and noticeable state differences.

EvoSuite[35] is the most relevant tool available for evolutionary mutation testing of Java based programs. EvoSuite can test large number of classes, packages, and projects but in every case, it tests a class in isolation hence it supports unit level testing of Java classes. EvoSuite is available free of cost online and testers can use it in two forms; through command line and as plug-in of Eclipse. EvoSuite supports only limited set of mutation operators [19 and 31] which are designed for structured paradigm. EvoSuite does not support any mutation operator, which can test an object-oriented feature for example method overriding, reference and content comparison, access modifier, and so on. EvoSuite uses various techniques for test data generation including hybrid search [32], dynamic symbolic execution [16], and testability transformation [15]. EvoSuite generates test cases for JUnit[40], which is a popular tool to perform unit testing.

Papadakis and Malevris [36] propose a control-oriented and state-based fitness function for evolutionary mutation testing. Their fitness function assigns four different costs to a test case. The first two costs their fitness function assigns are approach level and branch distance. Both of these costs evaluate as 0 if the mutated statement is executed and similar to the reachability cost. The third cost is predicate mutation distance that requires mutation distance, which is more or less same as of branch distance of failing condition. The impact distance (fourth cost) the fitness function assigns is similar to sufficiency cost to satisfy sufficiency condition. The fitness function calculates this cost the same way as proposed by Fraser and Zeller [34].

Fraser and Arcuri [39] present a new scalable approach to generate test cases for the programs. In this work they apply new optimization approaches to reduce the total effort, which is required for mutation testing. Using these optimization approaches first they monitor the state infection conditions to avoid redundant execution of test cases to kill mutants and secondly, instead of generating test cases for individual mutants, they do it for all the targets all at once to save time. In this research they have performed extensive experiments using the EvoSuite [35] tool. For experimentation, they have extended the tool to support the new optimization methods. Their experiments show good results and the results indicate that optimizations can help reducing mutation testing effort to a great deal.

IV. State-based & Control-oriented Fitness Function

We have proposed a new fitness function for evolutionary mutation testing in our previous work [38]. The main focus of the work is to use object's state and control flow information in evaluation of a test case. Earlier in the discussion (Section I) we have explained the importance of these features and with their inclusion, the testing process improves its performance in terms of time and becomes more effective. Search based algorithms like genetic algorithm can suffer from object's state problem [27] and the search process can become random if object does not gain desired state. A test case contains call to some methods of the class under test to bring it into a state after which, it can invoke the method to test the required feature. Object's state fitness tells the search process if the test case has gained the desired state or if it requires some more method invocations. This information further helps satisfying reachability and necessity conditions.

The second new element in fitness we propose using is control flow information along with output (data states) to evaluate sufficiency cost. In mutant program, mutated statement usually causes some change that should result in some deviation in flow of control as compared to original program. Eventually, this deviation in flow of control often produces different output. But if mutated program follows different path but produces same output, then it can be due to some logical programming error in the code by the programmer, which needs correction. We call such a mutant as suspicious mutant [36] and detection of such a mutant can help increasing mutation score after correcting the logical error in the program. Fraser and Zeller [28] also use method's statement coverage in fitness functions but their approach fails to exploit the strength of control flow information up to this potential.

In evolutionary mutation testing, a test case requires to satisfy three conditions to kill a mutant. These conditions include; reachability condition, necessity condition, and sufficiency condition. After including object's state fitness in first two conditions, we have given new names to them; state-based reachability condition and state-based necessity condition. Similarly, after including control flow information in sufficiency condition, we have given it new name as control-oriented sufficiency cost. In figure 2, we provide a code example to explain the application of our state-based and control-oriented fitness function. The code snippet contains definition of a `Number` class that provides a default constructor. The `Number` class also provides a method

```
public class Number {
  public Number() { }
  public int square(int a, int b) {
  1. int result = -1;
  2. if( a<b ) {
  3.   result = a;
  4. } else if( a>b ) {
  5.   result = b;
     } else {
  6.   result = a;
     } //END if-else STATEMENT
  7. result = result * result;
  8. return result;
  } //END square() METHOD
} //END Number CLASS
```

Fig. 2. Code snippet of `Number` Java class

definition with name `square()` with two integer parameters and it returns another integer, which the method calculates by taking square of the larger input parameter.

In order to explain how these conditions apply on a program, consider the code snippet in figure 2 again and assume statement 5 in the `square()` method contains mutation. Instead of directly assigning the variable `'b'` an absolute value is calculated using `Math.abs(b)` and then it is assigned to the `result` variable. The figure 3 presents control flow graph of the `square()` method and shows how three conditions apply there.

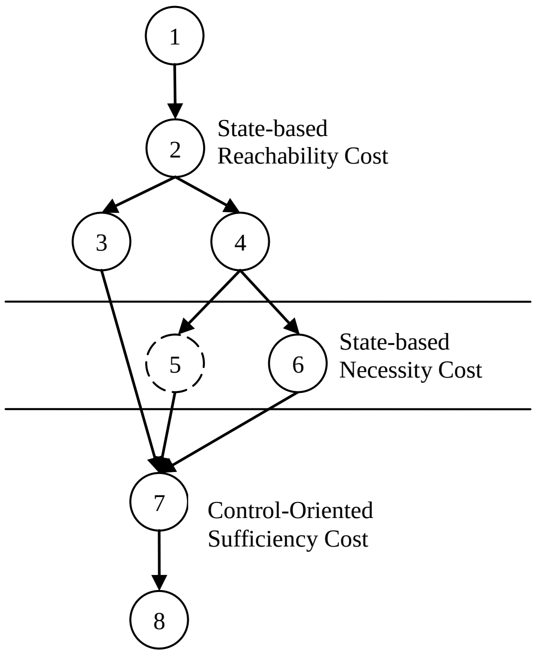

1
2
State-based Reachability Cost
3
4
5
6
State-based Necessity Cost
7
Control-Oriented Sufficiency Cost
8

Fig. 3. Control flow graph of `Number.square()` method

Two straight lines are dividing the control flow graph in three segments whereas each segment shows the cost associated to its corresponding condition. State-based reachability cost is calculated if a test case fails to reach the statement that contains mutation, otherwise fitness function assigns `'0'` to it. The fitness function calculates state-based necessity cost if a test case although executes the mutated statement but fails to introduce any change. Finally, fitness function calculates control-oriented sufficiency cost if a test case executes mutated statement and introduces a change in the program but it fails to reflect that change in the output of the mutated program.

Now we provide details about all three costs and method that our proposed fitness function adapts to calculate them. Each cost contains sub-costs that highlight problems in different parts of the test case.

A. *State-based Reachability Cost*

The fitness function calculates and assigns state-based reachability cost (SbRC) to a test case if it fails to execute the mutated statement. This cost comprises of two sub-costs including `state_fitness` and `coverage_fitness`. This cost is calculated at the point where control diverges away from the path that contains the mutated statement. The state-based reachability cost of a test case 't' is evaluated as follows:

SbRC(t) = (state_fitness, coverage_fitness)

Here SbRC(t) represents State-based Reachability Cost of test case 't' that has two main segments; `state_fitness` and coverage fitness. The `state_fitness` is further elaborated below;

state_fitness(t) = branch_distance(states)

The `state_fitness` indicates normalized branch distance of object's state variables that appear in the statement where control diverges away from the target. The object's state fitness is calculated only if the diverging point (usually if condition) contains state variable (data member of class) in predicate. If predicate returns true, it indicates that object is in desired state and hence object's state fitness is set as `'0'`. The existing approaches [6, 28, 33, 34, and 35] keep the branch distance of state variables with local variables due to which, the search process gets no information about object's state and this can cause the process to become random at times. The search process does not know which part of the test case is weak and needs improvement. The `coverage_fitness` is further divided into two sub-costs:

coverage_fitness(t) = [approach_level, branch_distance(vars)]

The `approach_level` is the difference between total number of dependent nodes and executed nodes and `branch_distance` is normalized branch distance of local variables involved. Both these values are calculated at the point where control diverges away from the target.

B. State-based Necessity Cost

The mutated statement should cause some change in the behavior of the program otherwise mutated program cannot produce a different output. In this case an appropriate cost needs to be assigned to the test case by fitness function. The existing techniques [6, 28, 33, 34, 35] uses branch distance to calculate necessity cost, which proves to be insufficient if mutated statement contains objects. Some object-oriented mutation operators make changes in object type through polymorphism, method overloading, method overriding and so on. So we need to consider object's state as well besides calculating only the branch distance for local variables. We propose adapting to the situation and use either of the following methods to calculate state-based necessity cost.

1) SbNC for Strcutured Mutation Operators

If conventional mutation operators like change in relational operator, change in arithmetic operator and so on are used then we propose using the normalized branch distance of the variables involved in the mutation including local variables, returned values of method or state variables of an object.

2) SbNC for Object-Oriented Mutation Operators

If a mutation is introduced using object-oriented mutation operators, for example call to a different (child) class constructor, change in method call and so on, then we first check if the mutation changes object's type. If object's type does change, we assign state-based necessity cost as `'0'` because a different object means difference in object's states hence a change is introduced. Otherwise if object's type remains the same, we propose calculating normalized branch distance on all the state variables of objects involved in original and mutated programs.

C. Control-oriented Sufficiency Cost

It is imperative for a change in the mutated program to be noticeable in the output otherwise original and mutated programs remain the same. The sufficiency cost deals with such a situation where mutant fails to reflect the change in output. In other techniques, we see an idea of comparing data states of original and mutant programs and fitness function penalizes the test cases for having equal data states [6].

We name our proposed sufficiency cost as Control-oriented Sufficiency Cost (CoSC). We propose comparing data states as well as control flow information while calculating sufficiency cost after the point where data states become equal. If the mutant program exercises different execution path than original program while the data states remain the same, we declare such a mutant as **suspicious mutant**. A suspicious mutant is one that produces similar output as of original program but its execution path is different that makes it suspicious. In some cases, mutant program does this because the program has some logical error, which masks the change, which is introduced by mutated statement. By removing these logical errors, it not only corrects the program in general but it also helps increasing mutation score.

D. Fitness Representation

After calculating all three costs, they are combined (not added) together to form complete fitness representation of a test case. The general format of test case fitness is shown as below:

fitness(t) = [SbRC(t); SbNC(t); CoSC(t)]

V. eMuJava

In this section we present the tool that implements our proposed fitness function and we name it as "**e**volutionary **Mu**tation Tool for **Java** Programs" or in short eMuJava. In our earlier work [38] the implementation was very limited in scope. In this research, we have extended the implementation to a great deal and have introduced new features in it to make it useable for experiments. We have used eMuJava to perform experiments (details are in section VI) to validate the effectiveness of our proposed fitness function. eMuJava has been implemented in Java programming language and is capable of testing Java programs eMuJava does not rely on any other third party or freeware component that makes it a complete and self-dependent automation. Figure 4 shows the architecture (module structure) of the tool having three large components and information being passed on to each of them by the environment and other components. Next we present inputs, outputs, and responsibilities of each component whereas later in this section we include some details on how eMuJava components co-ordinate and handle the assigned responsibilities.

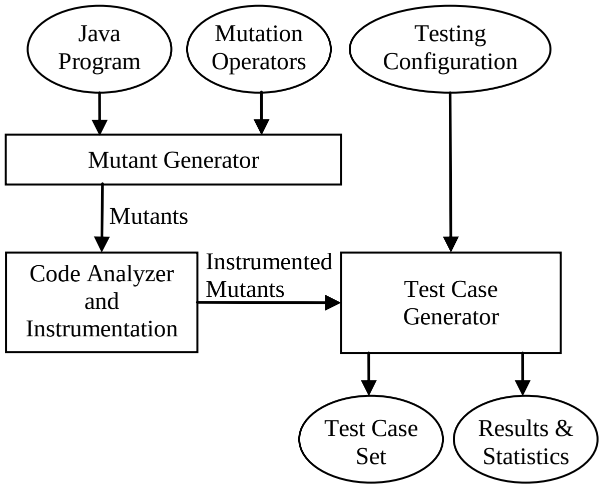


Fig. 4. eMuJava architecture

A. eMuJava Architecture

The implementation is complex in nature and based on three large components with a set of responsibilities assigned to each of them. Main inputs of eMuJava include Java program under test and mutation operators whereas after processing it produces test case set and results of experiment as result. Next we present inputs, outputs, and responsibilities of each component of the tool in detail.

1) Mutant Generator

The first component of tool is Mutant Generator from where the process of test case generation begins. It takes two inputs from the environment (usually provided by the tester) including source code to be tested and the mutation operators. User can load the source code in the tool by browsing through the directory structure of tester's computer and mutation operators can be selected by choosing them from the list provided on tool interface. The mutant generator then starts generating all possible mutants using the mutation operators. This process can take a while to complete because mutants can be large in number depending upon the Java classes under test and mutation operators selected by the tester. The output of this component is mutants in the form of Java classes containing exactly one fault per mutant.

2) Code Analyzer & Instrumentation

The output of Mutant Generator becomes the input of this component. This component is quite complex in nature as it comprises of some sub-components; including scanner, parser, class members (variables, constructors, methods) extractor, and the one which is responsible of performing instrumentation. Scanner and parser scan and parse the code to identify tokens from the code and to determine type of the statement they form. The class members extractor then merge the tokens and build larger set of groups for data members, constructors and methods. Finally, this component uses all the information to perform instrumentation by adding additional code within the statements to keep track of execution flow a test case will generate. This component generates instrumented Java code with no syntax error so it can be compiled and later executed. Finally, the instrumented mutants are compiled by this component.

3) Test Case Generator

This is the third and last component of eMuJava and it receives compiled instrumented mutants as input and starts the process of test case generation. eMuJava can use three different approaches for test case generation; random testing, standard genetic algorithm, and genetic algorithm with our proposed fitness function. This component receives the approach to be used for test case generation as input from the user along with other configuration details including number of test cases to be used as initial population, number of iterations to perform, crossover and mutation rate and so son. This component produces final test case set, results, and statistics of the experiment.

B. eMuJava Operations

This section explains how eMuJava performs various operations to conduct an experiment. The details of testing process from the input it receives until the output it produces.

1) Mutant Generation

Once eMuJava completes the process of identifying statements and tokens that form a given statement, it starts generating mutants. eMuJava generates mutants on the basis of mutation operators, the tester provides as input. eMuJava offers ten mutation operators such that five are structured and five are object-oriented mutation operators. The five mutation operators for structured paradigm are chosen from the work Offutt, Lee, Rothermel, Untch, and Zapf [4]. We have chosen these mutation operators because research [3] shows that the impact of applying these selective mutation operators is almost equal to applying the whole mutation operators set. The selective mutation operators include the following:

- Absolute value insertion (ABS)
- Arithmetic operator replacement (AOR)
- Logical connector replacement (LCR)
- Relational operator replacement (ROR)
- Unary operator insertion (UOI)

On the other hand the five object-oriented mutation operators are chosen from the research of Offutt, Ma, and Kwon [25]. We have chosen these operators such that we have a representation for every object-oriented feature:

- Overriding method deletion (IOD)
- `new` method call with child class type (PNC)
- Overloading method deletion (OMD)
- Member variable initialization deletion (JID)
- Reference comparison and content comparison replacement (EOC)

2) Population Generation

We have used the test case template proposed by Tonella [14] as shown in the figure 5. eMuJava begins the test case generation process, it generates initial population of solutions (test cases) randomly. Tester provides the size of the population as input whereas the default size is 50. For every given mutant, eMuJava looks for the method containing the

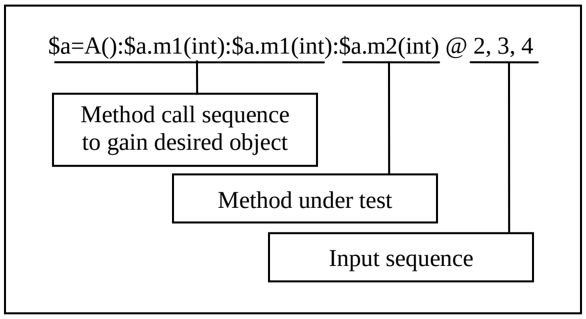

$a=A():$a.m1(int):$a.m1(int):$a.m2(int) @ 2, 3, 4
Method call sequence to gain desired object
Method under test
Input sequence

Fig. 5. Anatomy of a sample test-case for object-oriented program testing [28]

mutated statement and that method becomes the method under test.

The tool generates the test cases using the same template in which it creates an object of class under test followed by a random number of calls to some other method(s) of the same class and finally it calls to the method under test (method with mutation). Later the tool generates required input parameters randomly. eMuJava supports all the primitive data types of Java programming language as well as generation of `String` literals of random length. If the class under test depends on the object of another class then the tool generates the object of other class and makes call to some methods to gain the desired state.

Next we present details about three more operations that eMuJava performs including fitness function, crossover, and biological mutation. These are standard operations, which genetic algorithm performs; therefore, eMuJava does not perform them when tester chooses to use random testing for test case generation.

3) *Fitness Evaluation*

Our tool supports evaluates fitness of a test case using execution traces the test case generates. It evaluates fitness by two methods depending upon the approach tester chooses for test case generation. If tester chooses to generate test cases using standard genetic algorithm, then eMuJava uses fitness evaluation method of Bottaci [7]. On the other hand eMuJava uses our state-based and control-oriented fitness function for test case evaluation.

4) *Crossover*

On completion of the iteration, if the target (mutant) remains alive, the test cases are gone through crossover to generate off-springs for next iteration. eMuJava uses tournament selection method to select test cases for crossover, after which, pairs of test cases are formed. The tool uses single point crossover, which is performed on a randomly chosen point from the test cases. The following example illustrates how the single-point crossover is carried out by eMuJava:

Example
$a=A():a.m1(int):a.m2():a.mut(int,int) @2,4,5
$a=A():a.m2(int):a.m4(int):a.mut(int,int) @1,6,3,4

⇨ $a=A():a.m1(int):a.mut(int,int) @2,3,4
⇨ $a=A():a.m2(int):a.m4(int):a.m2():a.mut(int,int) @1,6,4,5

After the crossover, a new population of test cases is ready for next iteration.

5) *Biological Mutation*

The biological mutation is an important operation in genetic algorithm. It helps minimizing the chances of the search process getting stuck in local optima. After a specific number of iterations eMuJava performs biological mutation in the solutions. Mainly on a test case, the tool performs two types of mutations. If a test case has state fitness of `"0.0"` that means the object is already in a desired state and its method call sequence does not need a change. In this case eMuJava only mutates the input parameters of the method under test as shown in the following example:

Example
$a=A():a.m1(int):a.m2():a.mut(int,int) @2,4,5

⇨ $a=A():a.m1(int):a.m2():a.mut(int,int) @2,6,3

In other case if the object is not in the desired state (state fitness has a non-zero value) then the tool not only changes the input parameters of method under but it also changes the method call sequence. One or more method invocations may be introduced as well as one or more method invocation may be removed from the sequence as shown by the example below:

Example
$a=A():a.m1(int):a.m2():a.m3(int):a.m2():a.mut(int) @4,6,3

⇨ $a=A():a.m1(int):a.m2():a.m3(int):a.mut(int) @2,4,5

Once the mutation operation completes on all the test cases, new population is ready for the next iteration of genetic algorithm.

VI. Initial Experiment & Analysis

In this section we present results of initial experiments that we have performed using eMuJava to validate the effectiveness of the state-based and control-oriented fitness function (section IV). The experiment results are quite interesting and have helped identifying some limitations of standard genetic algorithm that we have discussed later in this section. Below we go in the detail, first we present some details about the platform we have used for experiments.

The experiments have been performed on Intel 32-bit machine having Core 2 Duo Centrino processor. The computer has got 2GB of memory (RAM) and enough hard disk to store the execution traces generated by the tool. The machine is running with Windows 7 operating system supporting 32-bit architecture. The tool has been run using Java virtual environment (JVM) version 8 with update 45 released by Sun Microsystems for Windows. Every time a new experiment is conducted the tool has been rerun to ensure the resources are completely released by JVM and new experiments are not affected by any means. We have also made sure that source code under test is error free and the experiments are not affected by any runtime exception generated by the tool.

We have chosen programs (containing one or more classes) from different domains for experiments and all of these programs are of different nature. Some of them are data structures (like `Stack` and `HashTable`), whereas others are popular programs (`Calculator` and `TemperatureConverter`) that we find in literature. Also we have chosen programs that automate home appliances (like `ElectricHeather` and `AutoDoor`). Table I presents list of all the programs that we have used for experiments. All the classes in programs exhibit variety of behaviors (through `public` methods) and in order to test those behaviors, some of the classes require their objects to be in specific state (for example `AutoDoor`, `Stack`, `ElectricHeater` and so on). On the other hand some of the methods contain mostly arithmetic expressions and conditional statements hence they do not have any serious requirement for objects to be in some specific state.

The experiments have been performed using eMuJava that we have implemented. eMuJava has been used to perform experiments with three approaches; first is random testing whereas second and third use genetic algorithm such that second uses standard fitness function and third one uses state-based and control-oriented fitness function. These approaches try to satisfy three conditions to kill a mutant; reachability, necessity, and sufficiency. Table I presents the results of the experiments in shape of average mutation score of multiple runs (ranges from 5 to 10) by each approach against all the programs. In order to check if our proposed fitness function guides the search process well and if it helps achieving high mutation score in less number of iterations, we did not allow the tool to run for indefinite amount of time. Instead we let the tool to execute 10 iterations per target to verify if it gains high mutation score with our proposed approach.

The results of the experiments have proven to be quite interesting. All the approaches have given good results in general but our proposed fitness function has produced a touch better set of test cases than the other two. For better understanding of the results, we have plotted them using column chart as shown in figure 6. In the chart, x-axis plots the tested programs (from Table I) and y-axis plots mutation score. Results achieved by approaches are represented with a different shade (random testing – light gray, GA with standard fitness function – dark gray, GA with proposed fitness function - black) to distinguish among them.

The proposed fitness function seems to obtain high mutation score for the programs that have state requirement

TABLE I. EXPERIMENT RESULTS

Programs	**LoC**	**M**	**Mutation Score (%)**		
			Random Testing	**GA with Standard Fitness Function**	**GA with Proposed Fitness Function**
AutoDoor	112	111	77	77	**80**
BankAccount	116	180	100	100	**100**
BinarySearchTree	119	189	87	85	**83**
Calculator	60	97	95	78	**83**
CGPACalc	105	111	96	84	**84**
ElectricHeater	120	146	87	84	**87**
HashTable	98	103	68	65	**74**
Stack	144	107	69	71	**88**
TempConverter	60	106	100	100	**100**
Triangle	94	147	79	80	**83**
Total	1028	1297	85.8	82.4	**86.2**

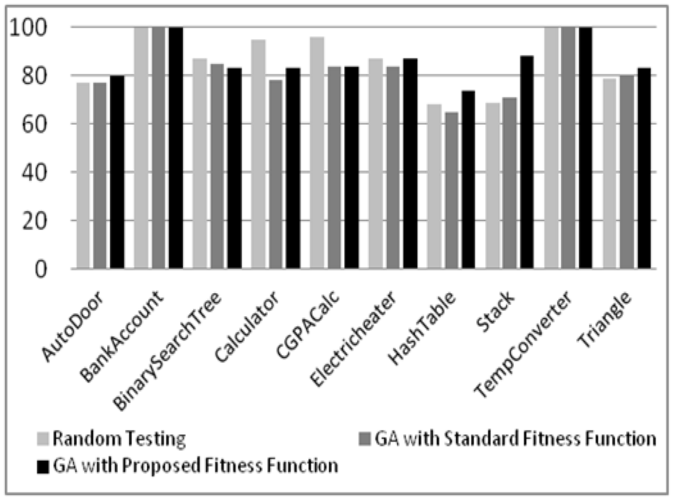


Fig. 6. Comparison of experiment results obtained from random testing, GA with standard fitness function, and GA with proposed fitness function

(`AutoDoor`, `HashTable`, `Stack`, and `Triangle`) but the difference in mutation score is not very significant. When we analyze the results, we discover that although the proposed fitness function provides information about the weakness in a test case (whether the object is in desired state through state fitness) yet the next phase of genetic algorithm (crossover) fails to utilize it due to its inherent nature. Crossover does not consider object's state fitness to produce new offsprings rather it randomly picks up a point in parents and crosses them over to form offsprings. So the search process has to wait until biological mutation to use state fitness to form new method call sequence, which may help object in gaining desired state.

Besides that for `ElectircHeater` our proposed fitness function manages equal mutation score and for `CGPACalc` it underperforms whereas both of these approaches have state requirement. Actually `ElectricHeater` and `CGPACalc` require values for state variables that do not have any particular pattern. Such values can be generated quickly with random generation rather than crossover of test cases. Similarly random testing produces better mutation scores for `BinarySearchTree` and `Calculator` as compared to genetic algorithm. `BinarySearchTree` and `Calculator` do not have state requirement but some predicates in the methods of these programs have strict argument requirement (for example `a==50`). In this scenario crossover cannot be useful and search process has to wait until biological mutation comes into play and produces required method arguments to satisfy the conditions in predicates.

If we can enable crossover to use the object's state fitness to produce offspring, we may be able to push mutation score further up in limited number of iterations. For this we have proposed a new crossover method that uses object's state information. Besides that if a program does not have state requirement and it relies on the method arguments, performing crossover for high number of iterations does not help. In this situation if we can dynamically increase rate of biological mutation, the search can produce required values quickly to kill the mutant in lesser number of iterations. We have proposed an adaptable mutation method for genetic algorithm that automatically adjusts the rate of mutation by assessing the situation using fitness of test cases. Both the proposals are presented in next section in detail.

VII. Two-way Crossover and Adaptable Mutation for Genetic Algorithm

In this section we present a new two-way crossover and adaptable mutation method for genetic algorithm that uses state fitness to produce and repair offsprings.

A. Two-way Crossover

To demonstrate how our proposed two-way crossover works and helps targets to diverge quickly, we have used example of `Stack` class.

Consider code snippet of `Stack` class in figure 7. The class has two private data members including `top` pointer and `elements[]` array to contain integers in stack formation. There are several methods defined in the class to simulate stack's behavior but we are interested in only `pop()` method definition for now. The `pop()` method pops out the top most element from the stack but there is an additional constraint on it. This method will remove and return the top element, only if there are at least 2 elements present in the stack.

Now we generate a mutant from `Stack` class such that the line 3 of `pop()` method contains a mutation. We want to generate a test case that will kill that mutant by satisfying all three conditions (reachability, necessity, and sufficiency). For a test case to kill this mutant, the object of `Stack` class is required to invoke `push(int)` method (its definition is not shown in figure 7) for at least 2 times before invoking `pop()` method otherwise test case will fail to execute the mutated statement. Suppose the genetic algorithm generates following two test cases as initial population;

Initial Population
T1: $s=Stack():s.push(int):s.mX():s.mY():s.pop() @ 22
T2: $s=Stack():s.mX():s.push(int):s.mZ():s.mA():s.pop() @ 6

Both the test cases cannot kill the mutant because they contain just one call to `push(int)` method hence object does not gain the desired state before calling `pop()`. Since both the test cases fail to achieve the target, fitness function comes into play for their evaluation. Our proposed fitness

```
public class Stack {
  private int top = 0;
  private int[] elements;
  …
  …
  public int pop() {
  1. int element = -1;
  2. if( top>=2 ) {
  3.   top = top - 1;
  4.   element = elements[ top ];
  5. } //END if STATEMENT
  6. return element;
  } //END pop() METHOD
  …
  …
} //END Stack CLASS
```

Fig. 7. Code snippet of `Stack` Java class

function will be able to highlight the discrepancy in them by assigning a non-zero value to state_fitness as shown below;

Fitness of Initial Population
Fitness(T1): [(**0.5**, [1, 0]) ; c ; (c, normal)]
Fitness(T2): [(**0.5**, [1, 0]) ; c ; (c, normal)]

The `0.5` fitness shows that object is not in desired state and method call sequence needs modification. After that, genetic algorithm performs crossover to form new population for next iteration. The one-point crossover picks a random point and bisects both the test cases on that. The new test cases are generated by merging the first part of first test case to the second part of second test case and similarly by merging first part of second test case and second part of first test case as shown below;

Current Population
T1: $s=Stack():s.push(int):s.mX():s.mY():s.pop() @ 22
T2: $s=Stack():s.mX():s.push(int):s.mZ():s.mA():s.pop() @ 6

New Population
T1: $s=Stack():s.push(int):s.mZ():s.mA():s.pop() @ 22
T2: $s=Stack():s.mX():s.push(int):s.mX():s.mY():s.pop() @ 6

If we carefully analyze the new population the new test cases have the same issue as the old test cases had. The reason of this problem is that the crossover does not consider the state fitness and does not try to repair test cases accordingly. To solve this problem, we propose a new method of performing crossover, which we call **two-way crossover** that takes into account object's state fitness to form new test cases.

In order to perform crossover, first we use tournament selection method to pick the good test cases from population. Then we form pairs of test cases from the selected test cases and crossover is performed on the pairs. In our proposed two-way crossover method, we merge both the segments of first test case to both the segments of second test case as shown below;

Current Population
T1: $s=Stack():s.push(int):s.mX():s.mY():s.pop() @ 22
T2: $s=Stack():s.mX():s.push(int):s.mZ():s.mA():s.pop() @ 6

New Population
T1: $s=Stack():s.push(int):s.mX():s.push(int):s.pop() @ 22, 6
T2: $s=Stack():s.push(int):s.mZ():s.mA():s.pop() @ 22
T3: $s=Stack():s.mZ(): s.mA():s.mX():s.mY():s.pop() @
T4: $s=Stack():s.mX():s.push(int):s.mX():s.mY():s.pop() @ 6

The two-way crossover doubles the test cases because it merges both the segments of each test case with both the segments of other. The new set of test cases contains a test case (T1) that gains desired object state before invoking `pop()` method. Similarly even strict state requirement can be fulfilled in next iteration by further crossing over test cases of this population. This `Stack` example has a simple state requirement but experiments have shown that the two-way crossover works equally well when situation demands even more complicated object's state.

The two-way crossover has tendency to generate more test cases for next iteration as compared to conventional crossover.

Every pair involved in crossover can generate up to four new test cases after performing two-way crossover. We adapt two ways to control this rapid increase in test cases. First we perform two-way crossover only on those test cases whose objects have state problem and have a non-zero state fitness. So if in a pair of test cases, both test cases have state fitness as zero, we do not perform two-way crossover on them. The reason is that if test case does not have state problem, this means the issue lies within the variable values passed as argument to method under test that needs modification and crossover cannot help here. Secondly, in worst-case scenario, if all the test cases of a given iteration have non-zero state fitness, then the next iteration will receive double amount of test cases and so on. To control this, when genetic algorithm performs biological mutation to ensure search does not get stuck in local optima, we reduce number of test cases back to the size of population, user sets in the beginning. By these two ways, we control the number of test cases during execution of genetic algorithm.

B. *Adaptable Mutation*

Mutation operation in genetic algorithm prevents the search process to get stuck in local optima. It is not possible to set one fix mutation rate that can produce best results as every program is different in nature and requires different type and range of input data. If mutation rate is too high, it will become close to random testing and if mutation rate is too low, the test cases will become similar after a certain number of iterations.

In evolutionary mutation testing, genetic algorithm can experience similar situation as shown by experiments in previous section. For applications like `BinarySearchTree` and `Calculator`, crossover cannot help as these applications require specific range of values that can be produced with random generations. But there are certain applications like `HashTable` and `Stack` that need some time to evolve test cases to satisfy complex predicates to kill the mutants. In this case high mutation rate can result in diverging the search away from its target. So an adaptable mutation rate is required to handle both scenarios.

Literature survey shows some research [17, 18, 26, 29, 30] has been done to provide basis to pick the right configuration for genetic algorithm and adapt mutation rate as per situation. Mainly these approaches decide about pre-maturity in population by analyzing the solutions (test cases) and comparing them whereas some approaches are specifically for structured paradigm. These approaches are not appropriate to help in object-oriented programs and for mutation based evolutionary testing.

In this case our state-based and control-oriented fitness function provides comprehensive information to genetic algorithm to decide if mutation rate needs an adjustment for next iteration or current configuration is fine. For understanding, consider the following general fitness of a test case '*t*';

fitness(t) = [SbRC(t); SbNC(t); CoSC(t)]
fitness(t) = [state_fitness(t), coverage_fitness(t); SbNC(t); CoSC(t)]

Once iteration completes, we propose analyzing fitness of all the test cases if the mutant (target) remains alive. If *state_fitness* of all the test cases remain "0.0", it shows test cases have gained required state or the program under test does not have state requirement. Now crossover of test cases cannot help because it does not modify arguments of method under test. In such a case increasing mutation frequency can save effort in producing required values for method under test. We propose decreasing interval between two mutations by "1" after unsuccessful iteration for programs that do not have state requirement or having "0.0" as *state_fitness*.

C. *eMuJava Version 2.0*

We have extended our eMuJava tool to provide support for proposed two-way crossover and adaptable mutation methods so we can perform experiments on it to test its effectiveness. We have named the new updated version as eMuJava version 2.0 or simply eMuJava v.2.

eMuJava v.2 now supports four approaches to generate test cases to kill the mutants. All these approaches generate test cases that can satisfy all three conditions to kill a mutant (reachability, necessity, and sufficiency). The approaches include random testing, genetic algorithm with standard fitness function, genetic algorithm with state-based & control-oriented fitness function, and the fourth one that has improved genetic algorithm with state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods.

eMuJava v.2 is an open source implementation and available free for download from URL http://jbillu.net/emujava.v.2/. The researchers and testers are welcome to download the tool and use it to perform experiments, extend the implementation and provide their valuable feedback to us if they want to propose a feature or to report a bug. The implementation is available in two forms; a complete project of NetBeans (Java IDE) that can be imported in NetBeans and secondly in the form of plain source code, which can be compiled and run manually through Java SDK.

eMuJava v.2 offers various new features to testers to increase usability and to save time. For example eMuJava v.2 provides testers the facility to view the generated mutants before proceeding to test case generation. Using mutant viewer, testers can view the mutation performed by the tool in a mutant and see if a given mutant is equivalent or non-equivalent. This helps performing correct calculation of mutation score. There is another very useful feature in the new version that allows the testers to filter equivalent mutants. Testers can list down all the equivalent mutants using mutant viewer feature of the tool and then filter them from the list before letting the tool to start generation test cases. The eMuJava ignore all those filtered mutants because they cannot be killed and hence it saves a lot of time.

VIII. Experiment Results

This section provides with results of extensive experiments carried out to validate the proposed two-way crossover and adaptable mutation methods. We have performed experiments using eMuJava v.2 on ten applications using four supported approaches by the tool. All four approaches generate test cases to kill the mutants and attempt to raise the mutation score. This section is further divided into three sub-sections and each section covers an aspect regarding experiments. We have performed experiments to show that our proposals help

TABLE II. RESULTS OF EXPERIMENTS PERFORMED USING EMUJAVA V.2

Application	Lines of Code	Mutants	Mutation Score (%)			
			Random Testing	GA with Standard fitness Function	GA with State-based & Control-oriented Fitness Function	Improved Genetic Algorithm
AutoDoor	112	111	77	77	80	**88**
BankAccount	116	180	100	100	100	**100**
BinarySearchTree	119	189	87	85	83	**88**
Calculator	60	97	95	78	83	**93**
CGPACalc	105	111	96	84	84	**100**
ElectricHeater	120	146	87	84	87	**90**
HashTable	98	103	68	65	74	**89**
Stack	144	107	69	71	88	**98**
TempConverter	60	106	100	100	100	**100**
Triangle	94	147	79	80	83	**89**
Total	1028	1297	85.8	82.4	86.2	**93.5**

generating a test case in less number of iterations, which can kill a mutant that requires object to be in a certain state. Secondly, we have compared the results with another tool EvoSuite, which is available for experiments. We have also shown with experiments that detecting **suspicious mutants** can help identifying a bug (logical programming mistake). By fixing such logical bug in the program, apparently looking equivalent mutant becomes non-equivalent that can be killed to raise mutation score.

A. Less Iterations High Mutation Score

First we present and discuss results of experiments that we have performed to validate if our proposed fitness function, two-way crossover, and adaptable mutation help the search process to diverge towards target in less number of iterations. We have performed experiments with eMuJava v.2 tool using four approaches to investigate this hypothesis. The experiments have been performed using same test configuration that includes number of iterations, number of test cases in a single iteration, crossover, and initial biological mutation rate. Before going into the details of experiments, below we present some information about four approaches we have used.

1. *Random Testing:* The test cases and input data are generated randomly in all iterations. If they fail to kill a mutant, the tool does not perform either crossover or mutation to repair the test cases rather it generates new population of test cases for next iteration.
2. *Genetic Algorithm with Standard Fitness Function:* This approach uses genetic algorithm to generate and evolve test cases. In first iteration test cases and input data are generated randomly that genetic algorithm later evolves in next iterations. If a mutant (target) remains alive after an attempt, eMuJava evaluates test cases using standard fitness function and then it performs single-point crossover and after certain number of iterations, it performs biological mutation to repair the test cases.
3. *Genetic Algorithm with State-based & Control-oriented Fitness Function:* This approach is identical to the approach we have discussed in 2) but the only difference it has is the fitness function. This approach implements our proposed state-based & control-oriented fitness function for evaluating test cases.
4. *Improved Genetic Algorithm:* This approach contains all of our proposed changes in the genetic algorithm including state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods.

We have presented results of experiments in Table II and they show that our proposed approach has performed well. For every approach we have used same test configuration and have not allowed the tool to run for indefinite amount of time. To kill a mutant, eMuJava runs 10 iterations each carrying population of 50 test cases. After every iteration fitter test cases are crossed over and after fifth iteration, test cases go through biological mutation. Random testing does not use crossover or biological mutation whereas our improved genetic algorithm uses two-way crossover and adaptable mutation rate. On completing 10 iterations, if mutant remains alive, tool leaves it and picks the next mutant.

If we carefully analyze the results, we notice that random testing and genetic algorithm with standard fitness function have not performed well on most of the programs specially those who have strict state requirement including `AutoDoor`, `HashTable`, `Stack` and so on. We have presented the comparison of these two approaches with genetic algorithm having state-based & control-oriented fitness function earlier (section VI). Now we compare our improved genetic algorithm with random testing and standard genetic algorithm. For better comparison we have plotted the results of these three approaches using a column chart in figure 8. All three are represented with distinguished colors (random testing - light-grey, genetic algorithm with standard fitness function - dark-grey, and improved genetic algorithm - black). The x-axis represents programs under test and y-axis represents achieved mutation score.

The results clearly show that our improved genetic algorithm has a significant impact on mutation score and it has increased for all those programs (`AutoDoor`, `CGPACalc`, `ElectricHeater`, `HashTable`, `Stack`, and `Triangle`) that have strict state requirement. It has obtained equal or touch higher mutation score for other programs (`BankAccount` and `TempConverter`) that do not have

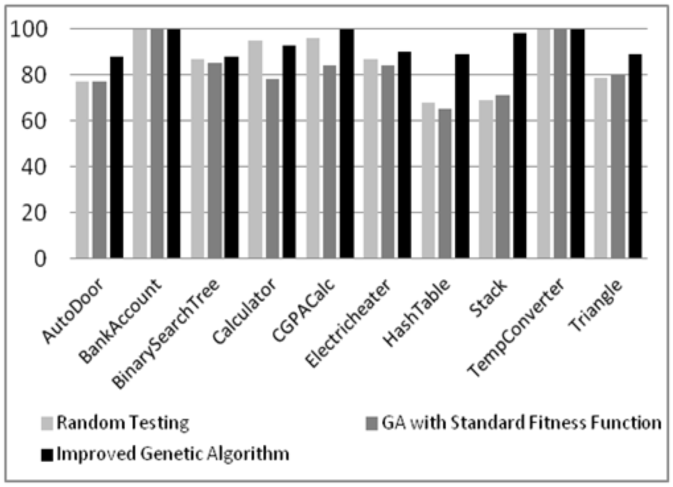


Fig. 8. Comparison of experiment results obtained from random testing, GA with standard fitness function, and our improved genetic algorithm

any state requirement. With the help of adaptable mutation, our proposed genetic algorithm has managed to complete random for programs (`BinarySearchTree` and `Calculator`) that require specific range of values. The difference in average mutation score obtained by our proposed approach from other approaches ranges approximately from 8% to 11%, which is promising.

We have analyzed our improved genetic algorithm with a different angle as well. Earlier we have presented comparison of mutation scores obtained by all the approaches in a fixed number of iterations. Now we shall see how many iterations all the approaches require to achieving 100% mutation score (after filtering out all the equivalent mutants). We have again performed experiments with eMuJava v2.0 tool and allow the tool to run until all the mutants get killed. The results are presented in Table III in shape of number of iterations consumed by each approach as well as number of test case generated by all of the four approaches

We have plotted the results with the help of line chart in figure 9. Figure 9 presents 10 line charts, one for each program, which has been tested. All four approaches are represented with four different colors (random testing - blue, genetic algorithm with standard fitness function - red, genetic algorithm with state-based & control-oriented fitness function - green, and our improved genetic algorithm - black). In these charts x-axis represents number of iterations whereas y-axis represents mutation score.

The line trace in these charts show the progress of all four approaches while they kill the mutants and achieve 100% mutation score. Our improved genetic algorithm seems to have clear advantage on all the other three approaches especially in the programs that have state requirement. For rest of the programs it performs equally well to the other approaches specially in comparison to random testing.

The results of these experiments prove our claim that with the help of state-based & control-oriented fitness function, the search process gets better guidance. Trough two-way crossover method test cases gain the desired state quickly. Adaptable mutation guides the genetic algorithm to increase the mutation frequency for programs that need specific values to satisfy conditions. Eventually improved genetic algorithm converges towards the targets in less number of iterations and helps in reducing required testing effort.

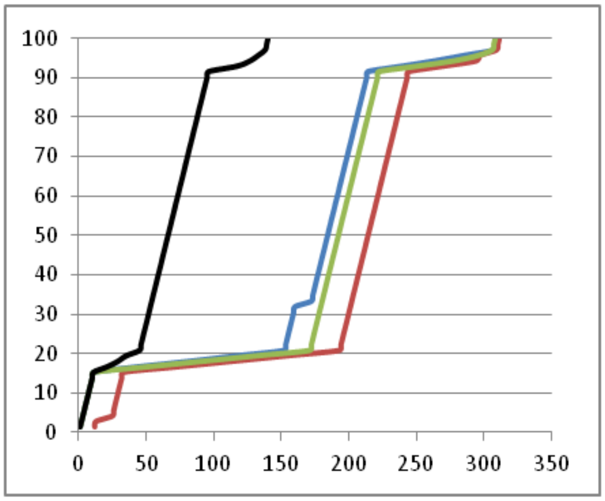
B. *Comparison with EvoSuite*

Now we present our experiment results in comparison to EvoSuite [30], which is the more relevant tool exists in the literature. EvoSuite can test Java based programs and can generate tests for branch coverage and mutation based coverage. It only supports unit level testing so EvoSuite cannot apply mutation operators that involve more than one class (mutation operators for inheritance or polymorphism). Also it does not support mutation operators that modify simple objects (access modifier related mutation operator and object vs. reference comparison and so on). In general EvoSuite does not support any object-oriented feature and uses limited set of conventional mutation operators [19 and 31] to generate mutants. Due to this reason although EvoSuite can test object-oriented programs in general but it can apply only structured mutation operators.

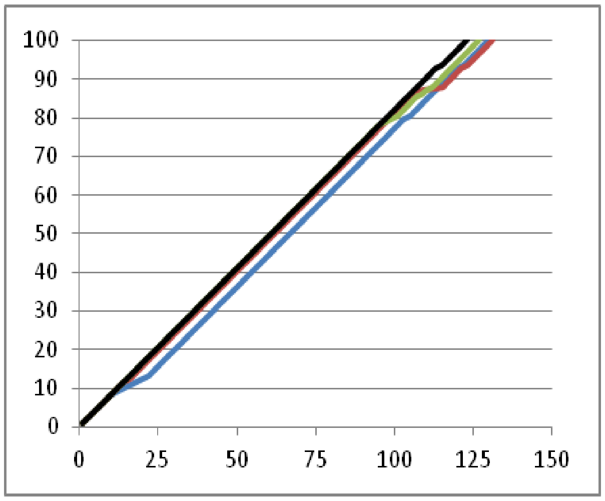
There are many issues that we have discovered while experimenting with EvoSuite. EvoSuite does not provide information about the mutants it generates, how many mutants remain alive, how does it handle equivalent mutants, and if it includes all the mutants in calculating mutation score. Also the

TABLE III. NUMBER OF ITERATIONS AND EXECUTED TEST CASES BY ALL FOUR APPROACHES

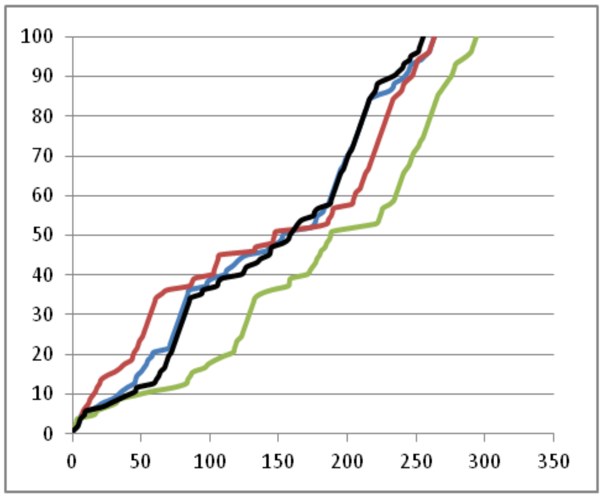
Programs	**Random Testing**		**GA with Standard fitness Function**		**GA with State-based & Control-oriented Fitness Function**		**Improved Genetic Algorithm**	
	Iterations	**Test Cases**	**Iterations**	**Test Cases**	**Iterations**	**Test Cases**	**Iterations**	**Test Cases**
AutoDoor	310	7750	311	7775	308	7700	**140**	**3500**
BankAccount	130	3250	131	3275	127	3175	**123**	**3075**
BinarySearchTree	263	6575	263	6575	294	7350	**255**	**6375**
Calculator	130	3250	280	7000	207	5175	**157**	**3925**
CGPACalc	93	2325	229	5725	195	4875	**74**	**1850**
ElectricHeater	380	9500	559	13975	589	14725	**375**	**9375**
HashTable	615	15375	569	14225	411	10275	**289**	**7225**
Stack	1052	26300	942	23550	450	11250	**208**	**5200**
TempConverter	91	2275	93	2325	87	2175	**83**	**2075**
Triangle	432	10800	476	11900	519	12975	**373**	**9325**
Total	3496	87400	3853	96325	3187	79675	**2077**	**51925**



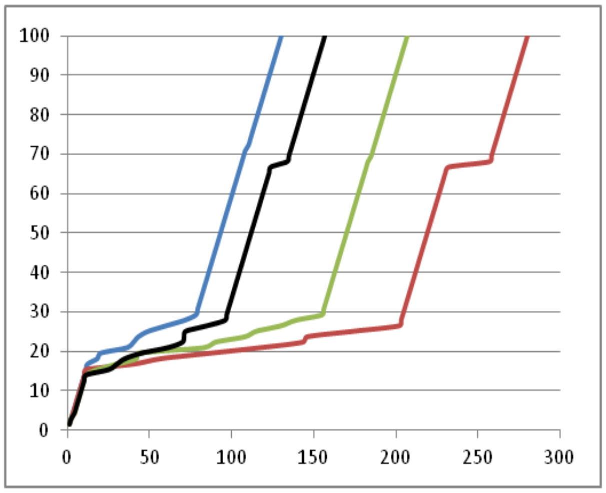
(a) AutoDoor



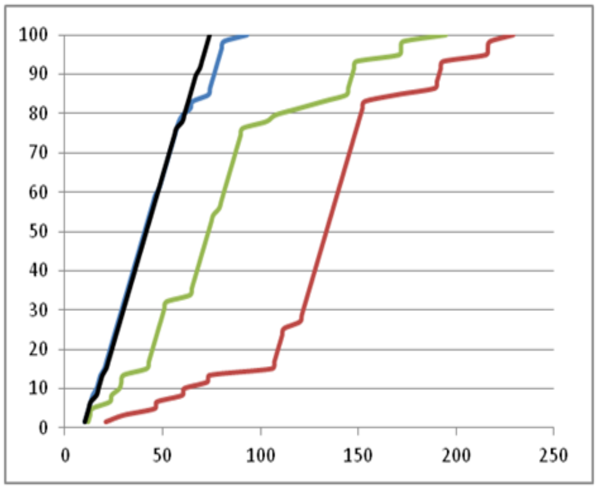
(b) BankAccount



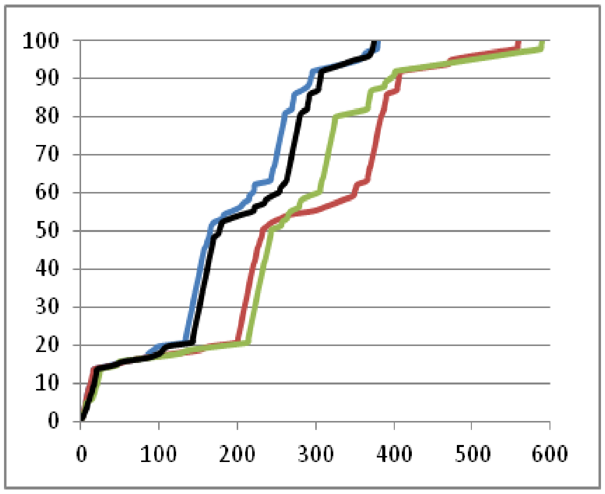
(c) BinarySearchTree



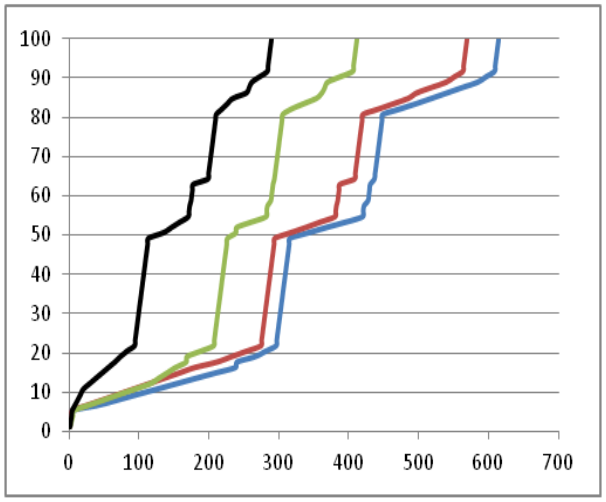
(d) Calculator



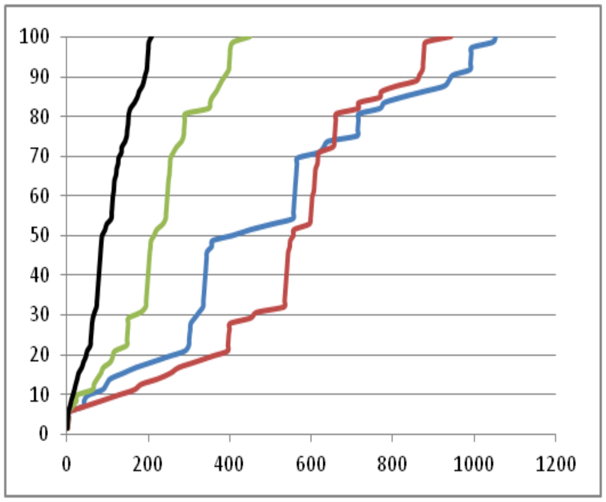
(e) CGPACalc



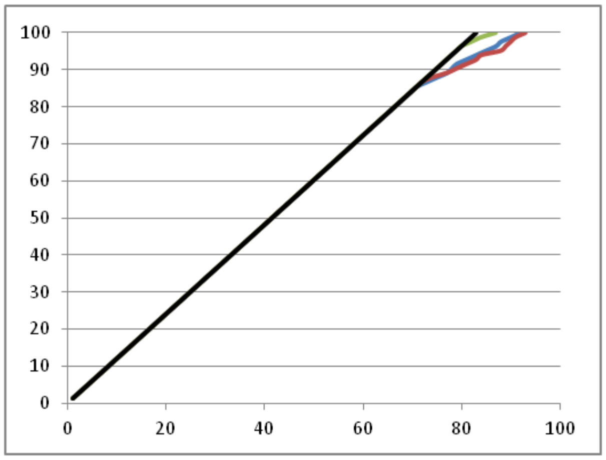
(f) ElectricHeater



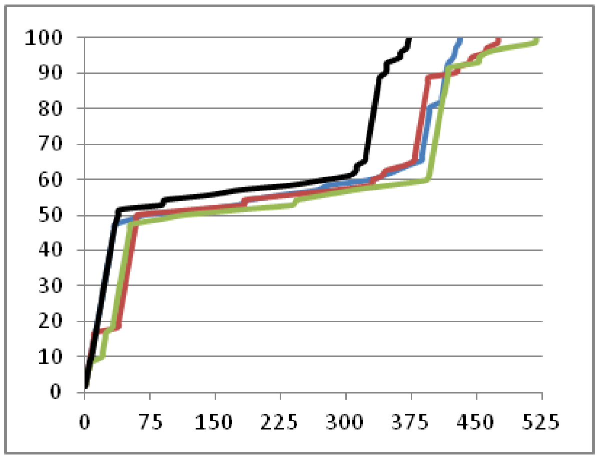
(g) HashTable



(h) Stack



(i) TempConverter



(j) Triangle

🟦 Random Testing 🟥 GA with Standard Fitness Function

🟩 GA with State-based & Control-oriented Fitness Function ⬛ Improved Genetic Algorithm

Fig. 9. Comparison to show progress among all approaches while achieving 100% mutation score

experiments we have performed earlier to compare our improved genetic algorithm with existing approaches, we restricted the iterations to see the performance of our approach but in case of EvoSuite it is not possible by any mean to limit the number of iterations to see how many targets it is able to achieve. Our implemented tool eMuJava differs a lot from EvoSuite and it is difficult to give an accurate comparison between them due to hidden information in EvoSuite. By making some adjustments and selecting the same mutation operators, we are able to produce a fair comparison between both approaches. We have presented the results of experiments in Table IV.

For better understanding, we have plotted the experiment results presented in figure 10 using column chart. The x-axis represents the programs we have used for experiments and y-axis shows average mutation score achieved after multiple runs. The grey bars represent mutation score achieved by EvoSuite whereas bars in black color represent mutation score achieved by eMuJava (our proposed approach). The difference between mutation scores shows that our improved genetic algorithm has performed well as compare to EvoSuite. eMuJava is able to kill more mutants in less number of iterations.

TABLE IV. EXPERIMENT RESULTS

Programs	**LoC**	**M**	**Mutation Score (%)**	
			EvoSuite	**Improved Genetic Algorithm**
AutoDoor	112	78	44	**82**
BankAccount	116	120	47	**95**
BinarySearchTree	119	102	38	**74**
Calculator	60	67	55	**86**
CGPACalc	105	59	59	**76**
ElectricHeater	120	103	29	**89**
HashTable	98	71	44	**74**
Stack	144	81	35	**85**
TempConverter	60	76	53	**95**
Triangle	94	72	50	**50**
Total	1028	829	45.4	**80.6**

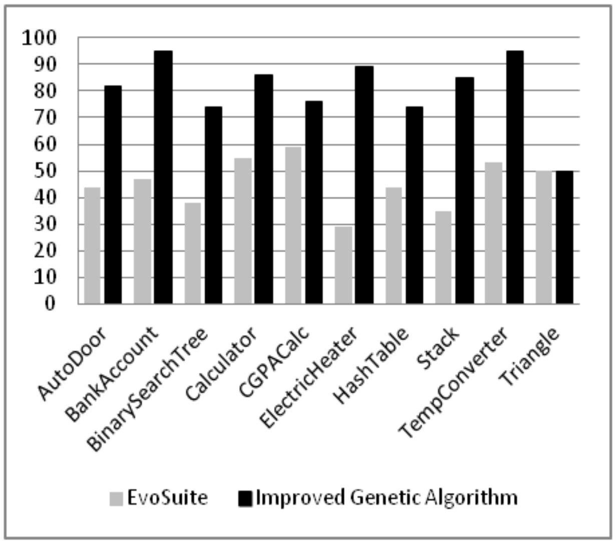


Fig. 10. Comparison of experiment results obtained from EvoSuite and eMuJava (our proposed genetic algorithm)

C. Detecting Suspicious Mutants to Raise Mutation Score

Sometimes, it has been noticed that a small programming mistake (logical) affects mutation score badly. Due to the logical error in program, the mutant becomes equivalent, which not only damages mutation score but it also wastes a lot of time during test case generation. In our earlier work [37], we coined an idea of using control-flow information besides output of the program to see how mutant behaves as compare to original program. This idea leads us to very interesting results and we noticed that if program has a logical error, although original and modified programs produce same output yet they exercise different execution path because sometimes the logical error masks the change introduced by mutation hence the mutant remains alive. We call such a mutant as **suspicious mutant**. So what our approach does is that, if a mutant remains alive but for a given test case, it exercises different execution path as compared to original program, we declare it suspicious and encourages manual checking of the program to ensure a logical programming error is not causing this problem. The figure 11 contains a code snippet of `CGPACalc` program that explains the concept of suspicious mutant;

The incorrect string literal at line 16 will always cause the method to return `"Pass"` even if the value of variable `cgpa` is less than `2.5`. Hence all the mutants generated from this method will remain alive. By using control follow information this error can be highlighted and correct, which results in raising the mutation score. We have used the same idea in our state-based & control-oriented fitness function that we have designed for evolutionary mutation testing of object-oriented programs (Section IV). In that work, we use control-flow information as part of test case fitness. Now in this research, we have performed experiments to prove its effectiveness.

```
  …
  public String calculateResult() {
    …
    //Mutated statement
    …
14. if( cgpa>=2.5 ) {
15.   return "Pass";
    } else {
16.   return "Pass";              // bug
    } //END if-else STATEMENT
  } /END calculateResult() METHOD
```

Fig. 11. `CGPACalc` mutant with logical bug (suspicious mutant)

None of the approaches including genetic algorithm, random testing, and EvoSuite uses control flow information as part of test case fitness, hence they suffer from such a situation where a logical error in the program causes the mutants to remain alive. To show the difference in mutation scores between our proposed work and others (genetic algorithm and EvoSuite), we have performed experiments using `CGPACalc` program by introducing the same error we have presented in figure 11. The figure 12 plots the results of experiments and proves the effectiveness of using control flow information to evaluate a test case.

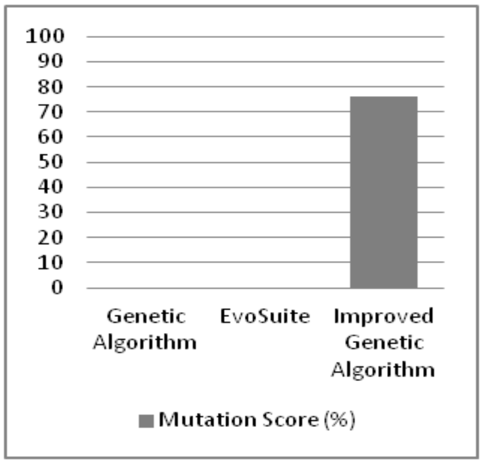


Fig. 12. Comparison of mutation scores obtained by genetic algorithm, EvoSuite, and improved genetic algorithm

IX. Conclusion and Future Work

Evolutionary mutation testing is a merger of two techniques; evolutionary testing and mutation testing. It is an effort to exploit strengths of evolutionary testing with combination of mutation testing to automate test case generation process and to reduce the computational overhead of mutation testing. Object's state plays a significant role in testing and control flow information also provides useful information about program's behavior. But according to our survey none of existing approaches use them. We have proposed a fitness function that uses object's state and control flow information as part of test case fitness. In this research, we have extended our work, implemented it in eMuJava tool, and performed experiments using it. The results have highlighted that although the object's state is made part of test case fitness yet this information is not used efficiently to improve the test case for the next iteration. On the basis of these results, we have proposed new two-way crossover and adaptable mutation methods, which are capable of using object's state fitness effectively to improve the test case. With these proposed changes the object gains the desired state quickly and methods receive required arguments. Eventually the test cases converge to the target in less number of iterations. We have extended the implementation of eMuJava to support two-way crossover and adaptable mutation in version 2. Using eMuJava v2.0 we have conducted extensive experiment. The experiments have given positive results and we are in the position to say that with the usage of object's state fitness, control flow information, two-way crossover, and adaptable mutation, the evolutionary mutation testing process improves. The testing process becomes effective and efficient because of the appropriate guidance the search process gets, mutation scores are increased, and logical programming errors are identified.

In future we plan to conduct experiments to compare our improved genetic algorithm with other search based techniques like particle swarm optimization (PSO) and artificial immune system (AIS). We also plan to apply our proposed modifications (state-based & control-oriented fitness function, two-way crossover, and adaptable mutation) on other evolutionary approaches to see if they work equally good as they do with the genetic algorithm. We plan to extend our eMuJava v.2 tool to support multiple evolutionary approaches to perform experiments and evaluation.

REFERENCES

[1] J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975

[2] R. A. DeMillo, A. J. Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Trans. Softw. Eng., 17, 9 (1991), 900-910.

[3] W. E. Wong, A. P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study", The Journal of Systems and Software, 31(3):185-196, December 1995.

[4] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, "An experimental determination of sufficient mutant operators," ACM Trans Software Eng Methodol 5, pp 99–118, 1996

[5] S. Kim, J. Clark, and J. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," In 12th International Conference Software & Systems Engineering and their Applications, December 1999.

[6] S. Kim, J. Clark, and J. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," In Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems, Erfurt, Germany, October. 2000

[7] L. Bottaci, "A genetic algorithm fitness function for mutation testing", In the Proceedings of International Workshop on Software Engineering using Metaheuristic Inovative Algorithms, a workshop at 23rd International Conference on Software Engineering, pp. 3–7, 2001.

[8] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing", Information and Software Technology, pp. 841-854, 2001.

[9] P. Chevalley, "Applying Mutation Analysis for Object Oriented Programs Using a Reflective Approach," In Proceedings of the 8th Asia-Pacific Software Engineering Conference, Macau SAR, China, December 2001

[10] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class Mutation Operators for Java," In Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering, pp 352-363, Annapolis MD, November 2002

[11] M. Bybro, "A Mutation Testing Tool for Java Programs," Thesis, Department of Numerical Analysis and Computer Science, Nada at the Royal Institute of Technology, KTH, Sweden, 2003.

[12] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia, "Mutation of Java objects," In Proceedings of IEEE 13th International Symposium on Software Reliability Engineering, 2003

[13] P. McMinn, M. Holcombe, "The state problem for evolutionary testing", In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724, pp. 2488-2497, Chicago, USA, 2003. Springer-Verlag.

[14] P. Tonella, “Evolutionary Testing of Classes”, In Proceedings of the ACM SIGSOFT International Symposium of Software Testing and Analysis, pp. 119-128, Boston, MA, July 2004.

[15] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation", IEEE Transactions on Software Engineering, 30(1):3-16, 2004.

[16] P. Godefroid, N. Klarlund, K. Sen, "DART: Directed Automated Random Testing", In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213-223, New York, USA, 2005.

[17] X. Xie, B. Xu, C. Nie, L. Shi, L. Xu, "Configuration Strategies for Evolutionary Testing", In the Proceedings of the 29th Annual International Computer Software and Applications Conference, 2005.

[18] B. Baudry, F. Fleurey, J.-M. Jezequel, Y. Le Traon, "Automatic Test Case Optimization: A Bacteriologic Algorithm", Published in IEEE Software, Volume: 22, Issue: 2, March-April 2005.

[19] J. H. Andrews , L. C. Briand , Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?", In the Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, May 15-21, 2005.

[20] M. Masud, A. Nayak, M. Zaman, N. Bansal, " A Strategy for Mutation Testing using Genetic Algorithms", IEEE CCECE/CCGEI, Saskatoon, May 2005.

[21] S. Wappler, F. Lammermann, “Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software”, In the Proceedings of The Genetic and Evolutionary Computation Conference, Washington, DC, USA, June 25-29, 2005.

[22] Y. Cheon, M. Y. Kim, A. Perumandla, “A Complete Automation of Unit Testing for Java Programs”, The 2005 International Conference on Software Engineering Research and Practice (SERP), Las Vegas, Nevada, USA, June 27-30, 2005.

[23] Y. Cheon, M. Kim, “A specification-based fitness function for evolutionary testing of object-oriented programs”, Proceedings of the 8th annual conference on Genetic and evolutionary computation, Seattle, Washington, USA, July 08-12, 2006.

[24] A. Seesing, H. Gross, “A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software”, ITSSA, pp. 127-134, 2006.

[25] J. Offutt, Y.S. Ma, and Y.R. Kwon. The class-level mutants of mujava. In AST ’06: Proceedings of the 2006 international workshop on Automation of software test, pp. 78–84, New York, NY, USA, 2006.

[26] C. S. S. Dharsana, A. Askarunisha, "Java Based Test Case Generation and Optimization Using Evolutionary Testing", In the Proceedings of International Conference on Computational Intelligence and Multimedia Applications, 2007.

[27] K. Liaskos, M. Roper, M. Wood, “Investigating data-flow coverage of classes using evolutionary algorithms”, Proceedings of the 9th annual conference on Genetic and evolutionary computation, July 07-11, London, England, 2007.

[28] M. B. Bashir, A. Nadeem, “A State based Fitness Function for Evolutionary Testing of Object-Oriented Programs”. In the Proceedings of 7th ACIS International Conference on Software Engineering Research, Management and Applications, Haikou, Hainan Island, China, December 2-4, 2009.

[29] I. Alsmadi, Using Genetic Algorithms for Test Case Generation and Selection Optimization", In the Proceedings of 23rd Canadian Conference on Electrical and Computer Engineering, 2010.

[30] M. Wang, B. Li, Z. Wang, X. Xie, "An Optimization Strategy for Evolutionary Testing Based on Cataclysm", In the proceedings of 34th Annual Computer Software and Applications Conference Workshops, 2010.

[31] D. Schuler, A. Zeller , "(Un-)Covering Equivalent Mutants", In the Proceedings of 3rd International Conference on Software Testing Verification and Validation, pp. 45-54 , 2010.

[32] M. Harman, P. McMinn, "A Theoretical and Empirical Study of Search based Testing: Local, Global and Hybrid Search", IEEE Transactions on Software Engineering, 36(2):226-247, 2010.

[33] K. K. Mishra, S. Tiwari, A. Kumar, A.K. Misra, "An Approach for Mutation Testing using Elitist Genetic Algorithm", In the proceedings of 3rd IEEE International Conference on Computer Science and Information Technology, pp. 426-429, Chengdu, China, 2010.

[34] G. Fraser, A. Zeller, "Mutation-driven generation of unit tests and oracles", In the Proceedings of the 19th international symposium on Software testing and analysis, pp. 147-158, 2010.

[35] G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416-419, New York, NY, USA, 2011.
[36] M. Papadakis, N. Malevris, "Automatic mutation based test data generation", In the Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, (GECCO '11), pp. 247-248, 2011.
[37] M. B. Bashir, A. Nadeem, "Control Oriented Mutation Testing for Detection of Potential Software Bugs", In the Proceedings of 10th International Conference on Frontiers of Information Technology, Islamabad, Pakistan, 2012.
[38] M. B. Bashir, A. Nadeem, “A Fitness Function for the Evolutionary Mutation Testing of Object-Oriented Programs”, In the Proceedings of the 9th International Conference on Emerging Technolgoies, Islamabad, Pakistan, 2013.
[39] G. Fraser, A. Arcuri, ”Achieving Scalable Mutation-based Generation of Whole Test Suites”, Empirical Software Engineering, Volume 20, Issue 3, 2015.
[40] Sourceforge, “JUnit”, http://www.junit.org/, accessed February 2017.