Abstract— Mutation Testing is used as fault-based testing to overcome limitations of other testing approaches but it is recognized as expensive process. In mutation testing, a good test case is one that kills one or more mutants, by producing different mutant output from the original program. Evolutionary algorithms have been proved its suitability for reducing the cost of data generation in different testing methodologies. In order to reduce the cost of mutation testing, efficient test cases are generated that reveal faults and kill mutants. In this paper, we develop a new strategy for generating efficient test input data in the context of mutation testing.

Keywords- Mutation Testing, Genetic Algorithm, Elitist Genetic Algorithm.

I. INTRODUCTION

The goal of testing is to uncover as many faults as possible with a set of test sets. Software organizations spend more than 40%-50% of their development cost in software testing. In order to test software, test data have to be generated. Generating test data manually is slow, expensive, and requires exhaustive efforts. So, automated test data generation techniques can be used to ease the process [4] and reduce the cost.

Testing can be of two types: black box testing and white box testing. Black box testing is mainly a validation technique that checks to see if the product meets the customer requirements. However, white box testing is a verification technique which uses the source code to guide the selection of data [3]. Mutation testing is a kind of white box testing. Basically, it is fault based testing based on mutation analysis which overcomes the limitations of other testing approaches. Mutation analysis identifies technique to mutate, i.e. to modify, software artifacts [2]. Mutation testing provides a testing criterion which can be used to measure the effectiveness of a test set or data in terms of its ability to detect faults.

Genetic algorithms are used to generate test data automatically which save the time and cost. But still there is a need to use genetic algorithm effectively to get results in less number of iterations which is possible only when minimal number of test cases that kills the maximum number of mutants are selected. We have modified simple genetic algorithm to achieve these goals. One drawback of simple genetic algorithm is related to initial population. In simple genetic algorithm initial population is generated randomly so searching of good test cases in minimum generation will be difficult. In order to motivate the generation of good test cases, we start with those test cases which have already killed some mutants in this way we can generate good test cases in minimum generations. Also to retain good test cases obtained in various generation we use elitist genetic algorithm.

The remainder of the paper is divided into following section: section 2 describes a brief introduction of mutation testing. Section 3 describes the overview of genetic algorithms. Section 4 describes the proposed approach followed by conclusion in section 5.

II. MUTATION TESTING

Fault based testing techniques: mutation analysis and mutation testing are often used to overcome the limitations of other testing approaches. Mutation analysis is used to identify mutation techniques while mutation testing tests adequacy criteria based on mutation analysis [2]. The history of mutation testing can be traced back to 1971 in a student paper by Richard Lipton [12]. The birth of mutation testing can also be identified in literature [16]. It can be applied at different levels: unit level, integration level, and specification level. Mutation testing is effectively used to identifying the adequate test data which can be used to find the faults [1]. It is also empirically proved that mutation testing is more powerful than the statement and branch coverage [15]. Mutation testing is more successful in finding faults than data flow testing [9, 10].

Mutation analysis is useful to assess and compare test suites and criteria in terms of their cost effectiveness. It can also be used to compare and assess new testing techniques [7]. The act of assessing test adequacy using mutation is referred as mutation analysis.

In mutation testing, faults are deliberately instrumented into the original program by making few syntactic changes in original program. The modified program is called as mutant. Similarly, several mutated copies of original program are generated. If the mutant and the original program generate different outputs for a test case then the mutant is called killed. If no test case can distinguish between the mutant and the original program then it is said to be alive [4]. There are some mutants that can never be killed, as they always produce the same output as the original program. These mutants are syntactically different but functionally equivalent to the original program [2]. These mutants are called as equivalent mutants. A good test case is one that kills one or more mutants, by producing different mutant output from the original program.
For an instance, consider following program fragment P,

1. if (x > 0)
2. doThis();
3. if (x > 10)
4. doThat();

Some of the possible mutants of P would be:

M1: 1. if (x < 0)
2. doThis();
3. if (x > 10)
4. doThat();

M2: 1. if (x == 0)
2. doThis();
3. if (x > 10)
4. doThat();

M3: 1. if (x > 0)
2. doThis();
3. if (x < 10)
4. doThat();

M4: 1. if (x > 0)
2. doThis();
3. if (x == 10)
4. doThat();

Here M1, M2, M3, and M4 are possible mutants of original program fragment P. All of these are first-order mutants as only single change is made to create every possible mutant.

Let D be the input domain and T be the test case for execution then T is a subset of D. To get the measure of mutation adequacy, every test case is run against each mutated copy of the original program and the number of mutants killed by every test case is counted (i.e. where T fails for a certain mutant). Mutants which are not killed by T are called alive. A mutated copy of original program is said to be equivalent: for every x belongs to D, such that, P.x is equal to M.x then the mutant M is said to be equivalent. We will also write it as M = P. Here P.x is evaluation of program P on the input x and M.x is the evaluation of program P on input x [5]. Therefore, mutation adequacy criteria or mutation score is defined as,

\[
\text{M Score}(T) = 100\times \left( \frac{d}{N} - \text{eq} \right)
\]  

(1)

Here ‘d’ is the number of mutants killed by the test case T, ‘N’ is the total number of mutants generated and ‘eq’ is the number of equivalent mutants i.e. M = P. However equivalent mutant’s problem is beyond the scope of this paper.

III. GENETIC ALGORITHMS

Genetic Algorithms (GAs) are adaptive heuristic search algorithm premised on the evolutionary ideas of natural selection and genetic [13]. The basic concept of GAs is designed to simulate processes in natural system necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of survival of the fittest. As such they represent an intelligent exploitation of a random search within a defined search space to solve a problem [6].

A genetic algorithm (GA) is a search technique used in computing to find exact or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. Genetic algorithms are a particular class of evolutionary algorithms (EA) that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination).

In software testing, our basic goal is to search the domain for input variables which meets with the testing objective. So, the problem is to maximize a function such that f(x1, x2, x3, ..., xm) and x1, x2,...,xm are variables to be adjusted towards global optimum[11]. Genetic algorithms are implemented in a computer simulation in which a population of abstract representations (called chromosomes or the genotype of the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

Genetic algorithm searches the input domain of the program in an order to generate suitable test cases to kill given mutant. It uses three operators to guide the search:

A. Selection

B. Crossover

C. Mutation

A variety of literature is available on this topic. Interested candidate may refer these references [14, 17].
IV. PROPOSED APPROACH

Our approach extends the work of Mattias Bybro[5] and M. Masud[4]. Mattias Bybro developed MU tool (Fig. 2) for java programming language. This tool automatically generates the mutated copies of the original program by instrument the mutants. Initially original program is tested by executing a set of test cases to check whether the program is correct or not. These set of test cases are generated by JUnit. Then, the meta mutants are generated using the operators given in Table 1 only if the original program is correct. Figure 1 elaborates the mutation testing process.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHF</td>
<td>Exception Handling Fault</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>LOR</td>
<td>Logical Operator Replacement</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator replacement</td>
</tr>
<tr>
<td>NOR</td>
<td>No Operation replacement</td>
</tr>
<tr>
<td>VCP</td>
<td>Variable and Constant Perturbation</td>
</tr>
<tr>
<td>MCR</td>
<td>Methods Call Replacement</td>
</tr>
<tr>
<td>RTI</td>
<td>Referencing Fault Insertion</td>
</tr>
</tbody>
</table>

TABLE 1 Mutation operators for OO program

M. Masud proposed a framework in which original program and mutant program are instrumented so that each program can be viewed as many small units. In his approach, instead of executing the entire program, every small unit is executed by the test case set in an order to kill the mutant unit.

M. Masud proposes that Genetic algorithm can be used to generate test cases for those mutants which are not killed by any chosen test cases. They have given no weightage to those test cases which have already killed some mutants.

Our approach extends the work of Masud’s by collecting all those test cases in a buffer which have already killed some mutants. These collected test cases will act as an
initial population of the genetic algorithm. In this way the probability of getting good test cases in less iteration can be increased. Moreover we use elitist version of genetic algorithm to collect various good test cases which are generated in different generation.

We use a checker module to compare and trace the output of each unit. The checker writes 1 in a log if the mutant unit survives and writes 0 if the mutant unit is dead. After running the test case on every program unit, if mutant is killed then that test case is selected as initial population because that test case have high probability to generate a good test case that kills the mutants in less iteration. In our approach, those test cases that killed some mutants in program unit are stored in a buffer whose size is equal to the initial population. When buffer is full then these test cases are used as initial population in an order to generate good test cases.

We also considered those test cases as initial population which reached specified mutation score threshold in an order to achieve the testing objective. Our approach is clearly illustrated in figure 4.

![Diagram of the proposed approach](image)

**Fig. 4 Our Proposed Approach**

V. CONCLUSION

In this paper, we proposed an approach through which good test cases are generated for mutation testing by using elitist version of genetic algorithm. It tries to find faulty unit in the code [4] as well as good test cases are generated having capability to kill the mutants. New test data are required to generate if a mutant survives after the execution of test case. To enhance the fault finding capability of a test case, new test cases are generated by taking those test cases as initial population which are capable of killing some mutants or whose mutation score is high. The proposed approach described in this paper is the extension of the work proposed by Mattias Bybro[5] and Masud[4]. However it is in preliminary stage. It requires some experimental comparison to claim its effectiveness. This approach can be used to develop a tool such as MU tool [5] in future.

**REFERENCES**


