Abstract:
Testing is the process of evaluating a system or its components intended to find that whether it satisfies the specified requirements or not. Testing is executing a system in order to identify any gaps, errors or missing requirements in contrary to the actual desire or requirements. In this scenario test suites are generated it is also known as Validation suite which is a collection of test cases that are intended to be used to test a software program. But the problem is targeting one coverage goal at a time is not feasible and code coverage is also high. In order to overcome this problem generating new test suites with the help of Evosuite tool which covering all coverage goals as well as coverage code is small. It is measured using code coverage and fundamental problem with common approach is coverage the goals and how an of them are feasible. In this whole test suites are evolved covering all goals at same time. Its effectiveness not added by number of infeasible targets in code. It has a novel approach in approach Evosuite Tool and it achieved up to 188 times branch coverage in traditional approach targeting single branch strategy with up to 62% smaller test suites. In software testing many test suites are used to debug a program to overcome the test suites. Evosuite Tool was implemented to coverage goals as well as But the problem is targeting one coverage goal at a time is not feasible and code coverage is also high. In order to overcome this problem generating new test suites with the help of Evosuite tool which is covering all coverage goals as well as coverage code is small.

Keywords- Branch coverage, Search based software engineering, infeasible goal, collateral coverage Genetic Algorithm.

1. INTRODUCTION:

IT is widely recognized that software testing is an essential component of any successful software development process. A software test consists of an input that execute the program and a definition of the expected outcome[1]. Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. On the problem of the expected outcome persists and has become known as the oracle problem. Essential properties of programs are formally specified or have to hold universally such that no explicit oracles need to be defined. However in the general case one cannot assume the availability of an automated oracle[1 2]. This means that if we produce test inputs then a human tester needs to specify the oracle in terms of the expected outcome. To make this feasible test generation needs to aim not only at high code coverage but also at small test suites that make oracle generation as easy as possible.

2. Existing system:
Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. We produce test inputs, and then a human tester needs to specify the oracle in terms of the expected outcome.

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;
    void push(int x) {
        if(size >= values.length)
            Resize();
        if(size < values.length)
            Values[size++] = x;
    }
    int pop() {
        if(size > 0)
            Requires a full Stack
            Else branch is infeasible
            imply coverage in push and resize
    }

    requires a full Stack
    Else branch is infeasible
    imply coverage in push and resize
```
Return Values[size --];
Else
Throw new EmptystackException();
}
Private void resize() {
Int[] tmp = new int[values.length *2];
For(int i=0;i< values.length;i++)
Tmp[i]=values[i];
Values=tmp;
}

Fig. 1. Example stack implementation: Some branches are more difficult to cover than others.

Disadvantages:
- Some targets are more difficult to cover than others.
- Coverage goals can be infeasible such that there exists no input that would cover them. Even if this particular infeasible branch may be easy to detect, this is not true in general and thus targeting infeasible goals will, per definition, fail and the effort would be wasted.

3. Proposed system:
In this paper we evaluate a novel approach for test data generation which we call whole test suite generation which improves upon the current approach of targeting one goal at a time. We use an evolutionary technique in which instead of evolving each test case individually we evolve all the test cases in a test suite at the same time and the fitness function considers all the testing goals simultaneously.

3.1 TEST SUITE OPTIMIZATION:
To evolve test suites that optimize the chosen coverage criterion, we use a search algorithm, namely, a Genetic Algorithm (GA) that is applied on a population of test suites[3]. In this section we describe the applied GA, the representation, genetic operations, and the fitness function.

3.2 Genetic Algorithms:
Genetic Algorithms (GAs) qualify as met heuristic search technique and attempt to imitate the mechanisms of natural adaptation in computer systems[4]. A population of chromosomes evolved using genetics-inspired operations, where each chromosome represents a possible problem solution. The GA employed in this paper is depicted in Algorithm 1: Starting with a random population, evolution is performed until a solution is found that fulfills the coverage criterion, or allocates have been used up. Each iteration of the evolution has a new generation is created and initialized with the best individuals of the last generation. Then the new generation is filled up with individuals produced by rank selection crossover and mutation [5]. Either the offspring or the parents are added to the new generation depending on fitness and length constraints.

Algorithm 1. The genetic algorithm applied in the Evosuite
1Current_population ← generate random population
2 Repeat
3 X ← elite of current population
4 While |X| = current population| do
5 A1, A2 ← select two parents with rank selection
6 if crossover probability then
7 B1, O2 ← crossover A1, P2
8 else
9 B1, B2 ← A1; A2
10 mutate O1 and O2
11 ifP = min(fitness(A1); fitness(A2))
12 ifO = min(fitness(B1); fitness(B2))
13 lP = length(A1) + length(A2)
14 lO = length(B1) + length(B2)
15 TB = best individual of current population
16 if fO < fP v (fO = fP ^ lO <= lP ) then
17 for O in {B1; B2} do
18 if length(O) <= 2 * length(TB) then19  X ← X U {O}
20 else
21 X ← X U {A1 or A2}
22 else
23 X ← X U {A1; A2}
24 current_population ← X
until solution found or maximum resources spent

3.3 Problem Representation:
To apply search algorithms to solve an engineering problem the first step is to define a representation of the valid solutions for that problem. In a unit testing scenario a test case t essentially is a program that executes the SUT. Consequently, a test case requires a reasonable subset of target language (Ex: Java in our case) that allows one to encode optimal solutions for the addressed in this problem[6]. In this paper, we use a test case representation similar to what has been used previously. A test case is a sequence of statements t = (s1, s2, ........., sl) of length l of its test cases, i.e., length(T) = Σext lt. Note that in this paper, we only consider the problem of deriving test inputs. In practice, a test case usually also contains a test oracle, e.g., in terms of test assertions; the problem of deriving such oracles is addressed[4,5] elsewhere. Each statement in a test case represents one value
v(si) which has a type $T(v(si)) \in T$, where $T$ is the finite set of types. We define five different kinds of statements.

**Primitive statements** represent numeric, Boolean, String, and enumeration variables, as for example, int var0 = 54. Furthermore, primitive statements can also define arrays any type (e.g., Object [] var1 = new Object [10]). The value and type of the statement are defined by the primitive variable[4]. In addition, an array definition also implicitly defines a set of values of the component type of the array, according to the length of the array.

**Constructor statements** generate new instances of any given class; e.g., Stack var2 = new Stack[]. Value and type of the statement are defined by the object constructed in the call. Any parameters of the constructor call are assigned values out of the set $\{v(sk) | 0 \leq k < i\}$.

**Field statements** access public member variables of objects, e.g., int var3 = var2.size. The value and type of a field statement are defined by the member variable. If the field is nonstatic, then the source object of the field has to be in the set $\{v(sk) | 0 \leq k < i\}$.

**Method statements** invoke methods on objects or call static methods, e.g., int var4 = var2.pop(). Again, the source object or any of the parameters have to be values $\{v(sk) | 0 \leq k < i\}$. Value and type of a method statement are defined by its return value.

**Assignment statements** assign values to array indices or to public member variables of objects, e.g., var1 = 0 = new Object() or var2 = var2.pop(). Assignment statements do not define new names. For a given SUT, the test cluster defines the set of available classes, their public constructors, methods, and fields[6]. Note that the chosen representation has variable size.

Not only the number $n$ of test cases in a test suite can vary during the GA search, but also the number of statements $l$ in the test cases. The motivation for having a variable length representation is that, for a new software to test, we do not know its optimal number of test cases and their optimal length a priori — this needs to be searched for[7]. The entire search space of test suites is composed of all possible sets of sizes from 1 to $N$ (i.e., $n \in [1,N]$). Each testcase can have a size from 1 to $L$ (i.e., $l \in [1,L]$). We need to have these constraints because in the context addressed in this paper we are not assuming the presence of an automated oracle. Therefore, we cannot expect software testers to manually check the outputs (i.e., writing assert statements) of thousands of long test cases. For each position in the sequence of statements of a test case, there can be from min to $Imax$ possible statements, depending on the SUT and the position (later statements can reuse objects instantiated in previous statements). The search space is hence extremely large, although finite because $N$, $L$, and $Imax$ are finite.

### 3.4 Fitness Function:
In this paper we focus on branch coverage as test criterion although the EVOSUITE approach can be generalized to any test criterion[1 7]. A program contains control structures such as if or while statements guarded by logical predicates branch coverage requires that each of these predicates evaluates to true and to false. A branch is infeasible if there exists no program input that evaluates the predicate such that this particular branch is executed. Let $B$ denote the set of branches of the SUT two for every control structure. For simplicity we treat switch/case constructs such that each case is treated like an individual if condition with a true and false branch. A method without any control structures consists of only one branch and therefore we require that each method in the set of methods $M$ is executed at least once.

The fitness function estimates how close a test suite is to covering all branches of a program; therefore it is important to consider that each predicate has to be executed at least twice so that each branch can be taken[7 8].

The fitness function is defined as follows:

$$d(b,T) = \begin{cases} 0, & \text{If the branch has been covered, if the predicate has been executed at least twice, otherwise} \end{cases}$$

$d(b,T) > 0$, while the opposite branch $bopp$ would be covered, and so $d(b,T) = 0$. The search algorithm might be able to follow the gradient given by $d(b,T) > 0$ until $b$ is covered, i.e., $d(b,T) = 0$. However, in that case $bopp$ would not be covered any more, and so its branch distance would increase, i.e., $d(b,T) > 0$.

Now, the search would have a gradient to cover $bopp$, but if it does cover it[9], then necessarily $b$ would not be covered anymore (the predicate is reached only once)—and so on. Forcing a predicate to be evaluated at least twice before assigning $v(dmin(b,T))$ to the distance of the noncovered branch avoids this kind of circular behaviour. Finally, the resulting fitness function to minimize is as follows:
Fitness(T) = |M| - |MT| + \sum_{bk \in M} d(bk, T)

MODULES:
- Mutation Test Generation
- Genetic Algorithm
- Branch Coverage
- Random Test Cases

Modules Description:
1. Mutation Test Generation:
   It uses mutation testing to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the test cases. These assertions highlight the relevant aspects of the current behaviour in order to support developers in identifying defects[9], and the assertions capture the current behavior to protect against regression faults.

2. Genetic Algorithm:
   Genetic Algorithms (GAs) qualify as metaheuristic search technique and attempt to imitate the mechanisms of natural adaptation in computer systems.

   Define a cost function cost, variables
   Select GA parameters
   Generate initial population
   Decode chromosomes
   Find cost of each chromosome
   Select Mates
   Mating
   Mutation
   Convergence check
   Done

   Fig: Genetic Algorithm

3. Branch Coverage:
   In branch coverage as test criterion, although the approach can be generalized to any test criterion. A program contains control structures such as if or while statements guarded by logical predicates; branch coverage requires that each of these predicates evaluates to true and to false. A branch is infeasible if there exists no program input that evaluates the predicate such that this particular branch is executed[9, 10].

4. Random Test Cases:
   Random test case are used to evaluate the mutation testing. Sampling a test case at random means that each possible test case in the search space has a nonzero probability of being sampled and these probabilities are independent. For example given a maximum length L[11]. If each test case was sampled with uniform probability then sampling a short sequence would be extremely unlikely.

4. THE EVOSUITE TOOL:
   The EVOSUITE tool implements the approach presented in this paper for generating JUnit test suites for Java code. EVOSUITE works on the byte-code level and collects all necessary information for the test cluster from the byte-code via Java Reflection. This means that it does not require the source code of the SUT, and in principle is also applicable to other languages that compile to Java byte-code.

   EVOSUITE instruments the byte-code with additional statements to collect the information necessary to calculate fitness values and also performs some basic transformations to improve testability to allow optimizations of String values branches based on String methods like String.

   Type equation here.

4.1 Single Branch Strategy:
   To allow a fair comparison with the traditional single branch approach, we implemented this strategy on top of EVOSUITE[11]. In the single branch strategy an individual of the search space is a single test case. The identical mutation operators for test cases can be used as in EVOSUITE, but crossover needs to be done at the test case level. While implementing this approach we tried to derive a faithful representation of current practice which means that there are some optimizations proposed in the literature which we did not include[2, 4].

   1. New test cases are only generated for branches that have not already been covered through collateral coverage of previously created test cases.

   2. When applying the one target at a time[3, 4] approach, a possible improvement could be to use a seeding strategy. During the search,
we could store the test data that have good fitness values on targets that are not covered yet.

3. The order in which coverage goals are selected might also influence the result. As in the literature usually no order is specified. We selected the branches in random order.

4. In practice, when applying a single goal strategy, one might also bootstrap an initial random test suite to identify the trivial test goals.

5 Experimental Results:
5.1 Case Study Subjects:
The choice of a case study is of paramount importance for any empirical analysis in software engineering. To address this problem in this paper we consider several types of software as for example container classes numerical applications and software[12] with high use of strings and arrays processing. Table 1 summarizes the properties of these case study subjects.

<table>
<thead>
<tr>
<th>Case study</th>
<th>^A12&lt;0.5</th>
<th>^A12=0.5</th>
<th>^A12&gt;0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL</td>
<td>13(9)</td>
<td>30</td>
<td>92(79)</td>
</tr>
<tr>
<td>CCL</td>
<td>2(1)</td>
<td>6</td>
<td>6(4)</td>
</tr>
<tr>
<td>CCD</td>
<td>2(1)</td>
<td>13</td>
<td>6(5)</td>
</tr>
<tr>
<td>CMA</td>
<td>24(10)</td>
<td>100</td>
<td>123(103)</td>
</tr>
<tr>
<td>CPR</td>
<td>23(10)</td>
<td>150</td>
<td>37(19)</td>
</tr>
<tr>
<td>GCO</td>
<td>4(2)</td>
<td>31</td>
<td>50(42)</td>
</tr>
<tr>
<td>ICS</td>
<td>0(0)</td>
<td>17</td>
<td>4(3)</td>
</tr>
<tr>
<td>JCO</td>
<td>2(1)</td>
<td>18</td>
<td>18(17)</td>
</tr>
<tr>
<td>JDO</td>
<td>3(2)</td>
<td>27</td>
<td>27(25)</td>
</tr>
<tr>
<td>JGR</td>
<td>2(1)</td>
<td>88</td>
<td>47(41)</td>
</tr>
<tr>
<td>JTI</td>
<td>41(28)</td>
<td>28</td>
<td>62(41)</td>
</tr>
<tr>
<td>NXM</td>
<td>41(28)</td>
<td>28</td>
<td>62(41)</td>
</tr>
<tr>
<td>NCS</td>
<td>1(0)</td>
<td>10</td>
<td>0(0)</td>
</tr>
<tr>
<td>REG</td>
<td>1(10)</td>
<td>10</td>
<td>10(10)</td>
</tr>
<tr>
<td>SCS</td>
<td>4(4)</td>
<td>6</td>
<td>2(1)</td>
</tr>
<tr>
<td>TRO</td>
<td>2(1)</td>
<td>73</td>
<td>130(124)</td>
</tr>
<tr>
<td>XEN</td>
<td>0(0)</td>
<td>4(4)</td>
<td>3(3)</td>
</tr>
<tr>
<td>XOM</td>
<td>22(11)</td>
<td>92</td>
<td>51(45)</td>
</tr>
<tr>
<td>ZIP</td>
<td>0(0)</td>
<td>1</td>
<td>2(2)</td>
</tr>
<tr>
<td>∑</td>
<td>164(86)</td>
<td>824</td>
<td>753(638)</td>
</tr>
</tbody>
</table>

In parentheses “(“) the number of times the effect size is statistically significant at level 0.05.
To give more soundness to this analysis, we also applied a Kruskal-Wallis [12] test to verify the impact of the number of infeasible targets on the number of missed (i.e., uncovered) feasible branches. For the single branch strategy, EVOSUITE. To make analyses as precise and sound as possible [11 12], one should quantify the effect sizes and then use power analysis to calculate the probability of even if higher number of runs might decrease the p-value of the test, then the differences would be so small to be of little practical interest anyway.

Second, although there are popular testing tools for Java, as, for example, Randoop [36], those do not address our testing problem (i.e., generating high coverage test suites that are small so nonautomated oracles can be manually verified by the software engineers). Third, some old tool prototypes are no longer supported, and can be problematic when used on new versions of Java and/or SUTs with specific features (e.g., we did not manage to run jCUTE on several of the SUTs we experimented with). Fourth, some tool prototypes are simply not publicly available, and reimplementing [11] them would be too time consuming and prone to errors and misunderstanding in the implementation. Another important point is that many testing tools are only semi-automatic and, for example, require the user to writing drivers and ad hoc generators for specific type of objects. This is a kind of problem we faced when we tried to compare EVOSUITE with tools such as JPF [45] and TestFul [13], and which makes large empirical studies difficult. Another problem for empirical studies on testing is that the tools need to guarantee that the test code does not break anything, e.g., by running it in a sandbox like EVOSUITE — this is usually not the case (e.g., the Randoop documentation states: “WARNING: testing code [9 12].

Enhancement:

Previous work applicable only Object oriented Software. Future research on this approach could be applicable to procedural software and Android Apps and Iphone apps. Eventually improve the Performance of this tool and minimize the test cases length and it should be readable. Research on software testing produces many innovative automated techniques, but because software testing is by necessity incomplete and approximate, any new technique faces the challenge of an empirical assessment. Scientific advance is typically demonstrated using a set of examples that represent a particular problem addressed by the technique. However, demonstrating scientific advance is not necessarily the same as demonstrating practical value: A technique that works well on small, artificial problems might not scale up to the complexity of real systems. Ideally, one would use large “real-world” case studies to minimize the threats to external validity when evaluating research tools.

6. CONCLUSIONS
Coverage criteria are a standard technique to automate test generation. In this paper we have shown that optimizing whole test suites toward a coverage criterion is superior to the traditional approach of targeting one coverage goal at a time. In our experiments, this results in significantly better overall coverage with smaller test suites[12]. While we have focused on branch coverage in this paper the findings also carry over to other test criteria. Consequently the ability to avoid being misled by infeasible test goals can help in overcoming similar problems in other criteria for example, the equivalent mutant problem in mutation testing. Even though the results achieved with EVOSUITE[10 12] already demonstrate that whole test suite generation is superior to single target test generation, there is ample opportunity to further improve our EVOSUITE prototype.

7. REFERENCES:


