Search-Based Junit Test Case Generation of Code Using Object Instances and Genetic Algorithm

Pranali Prakash Mahadik and D. M. Thakore

Computer Engineering Department, Bharati Vidyapeeth Deemed University College of Engineering Pune, Pune-411043, India
pranali72@gmail.com, dmthakore@bvucoep.edu.in

Abstract

To ensure software quality software testing is done. Test data generation is one of the more expensive parts of software testing. So, for reducing software cost and development time automation of test data generation is required. The object-oriented paradigm can be challenging for generating test data, due to some aspects of its features like abstract class, encapsulation, inheritance and visibility.

To reach high code coverage of object oriented code search based test data generation technique is used. Proposed approach show that how efficiency and effectiveness of search based test data generation using static analysis. System takes input as different java class files then instances are generated for that classes then generate sequence of method call for whole code coverage then using genetic algorithm which is very useful and work efficiently when there is large search space it use to reach test target and finally generate test cases in Junit format which are helpful for all developers to perform unit testing.

Keywords: Automatic test cases generation, unit class testing, search based testing, Junit, genetic algorithm, Object-oriented testing challenges

1. Introduction

Test-data generation is an essential and important stage of software development. Manually generating test-data is effort-consuming task. Revealing hidden errors in code implementation is very time consuming around 50% time of the total software life-cycle [1]. In software testing phase Test-data generation is most expensive parts. Therefore, automating this task can considerably reduce software cost and development time.

Testing object-oriented program is somewhat challenging task because of its features like encapsulation, inheritance, abstract class and visibility restrict direct access to some code part because of this whole test coverage is impossible. In practice test case generation is the most laborious and resource as well as time consuming process in software testing hence producing test data for object-oriented code is even more challenging and effort demanding.

To avoid this problem most of the approaches like symbolic execution, search based and random test data generation are proposed. Search Based Software Testing (SBST) name itself indicate coverage of code this approach effectively applied to resolve the problem of test-data generation [2], [3], [11]. For object oriented code for that it generates instances of class and call sequence of methods. SBST consider object state of class. Object-oriented testing aims at automating Junit test case generation process using evolutionary strategy like Genetic Algorithm. Evolutionary testing [28] helpful to the software testers to speed up the process and achieve reduction in the amount of project resources.
Propose search-based approach effectively used for object-oriented test-data generation and to addresses problem of accessibility. It also allows us to reach test target and generate Junit test cases proposed system compared with random testing approach. Using GA improves the search from one generation to the next, and gives better coverage than random testing. Another observation is that random testing produces less successful test cases than the proposed GA.

Main components of GA:

a. A chromosome is a representation of a feasible solution to the problem and each component of chromosome is a gene;
b. A population of encoded component;
c. A fitness function use to assign a score to each component;
d. A selection of n number of [46][47][48] operator to select parents according to their fitness;
e. A crossover operator recombines and generates new offspring;
f. A mutation operator differentiates the population by presenting new offspring.

In this approach we provide input as a class path and a Java source file or a directory. And produce Junit test cases [29] using instance generator of classes that is based on a generator of means-of-instantiation to reach better code coverage. To address the unit test case generation problem use genetic algorithm and produce Junit test cases which are very helpful for expert developers as well as for novice developer.

2. Related Work

Many approaches are available to generate automatic test cases which include different tools and techniques which summarized as follows,

2.1. Code based test data generation methods

There are mainly 3 methods used to generate test data,

2.1.1. Random Test Generation: Random test generation generates good result in test data generation. It can randomly select input until inputs are found [4]. Random test generation is very easy as well as fast method to generate test data. It can generate huge number of test cases automatically.

2.1.2. Symbolic Execution based techniques: Symbolic execution is a technique for test-data generation proposed by James King in 1976 [5]. Symbolic execution based technique contain dynamic symbolic execution and concolic testing. Symbolic test data generation techniques [6, 7] allocate symbolic values to the variables and produce algebraic expressions for the many constraints in the program code. A constraint solver is used to determine solution for these expressions that cover a test requirement. Symbolic execution can be used for numerous purposes, like as detection of bug, program verification, debugging of code, maintenance, and localization of fault [8].

2.1.3. Search-based techniques: Search algorithm name itself represents some kind of coverage. it has considerable applications in test data generation because the producing software tests is an undesirable problem [9,10]. It use as optimization technique for test data generation and to solve software engineering problems. Benefit of using SBST is that its outcome shows the efficiency of approach only drawback is it requires large search space.
2.2. Analysis of Various Automatic Test Generation Tools for Object Oriented Code

There is some automated test generations tools are available for object oriented code depending upon literature survey we collected all the automation supported unit-testing tools for object oriented code. Then categorize them by using some parameters in Table 1.

3. Challenges to Testing Object-Oriented systems

Object oriented code features generate different challenges while generating test data summarized as follows,

3.1. Encapsulation/Data Abstraction

Encapsulation means wrapping up of data and functions into a single unit called class. Class is combination of both data and functions. Data is not accessible from the outside world and only those functions which are present inside the class can access the data. The prevention of direct access of data by the program is called data hiding. Hiding the complexity of program is called Abstraction and only essential features are represented i.e., internal working is hidden. In the encapsulation observation of state of an object is done through its operations; therefore a fundamental problem of observability along with controllability [31].

3.2. Inheritance

Inheritance is process by which object of one class can acquire the properties of object of another class. Inheritance generates the issue of retesting. Here methods inherited from ancestor should retest in descendant class.

Inheritance generates issues from testing point of view like,

a. We need to test all features of classes we inherit from parent class again in derived class.

b. Need to test whether a subclass specific constructor is correctly invoking constructor of the base class because base class constructor executed first when both derived and base class having constructor.

c. In inheritance change made in super class affect on subclass so we need to retest all its subclasses.
d. More the depth of a tree means more dependency among the classes here all derived classes need to be tested again who are inheriting the features of base class.

e. In multiple inheritance derived class inherit properties from more than one base class suppose two super classes having same function it will create ambiguity which code is to be used.

f. If base class having private members then it will create issue for accessing it in derived classes.

3.3. Polymorphism

Polymorphism is mechanism is ability to take more than one form it allow object having different internal structure to share the same internal interface. Also an attribute of an object may refer to more than one type of data, and a function may have more than one implementation which leads to lack of controllability as actual binding of object reference is not known till run time also lead to lack of controllability. Polymorphism has two types,

1. Static binding - function calls can be resolved at compile time as in procedural language procedure calls are bound statically [32].

2. Dynamic binding - the decision to which method is to be used is done at run time. Hence dynamic binding is closely related to Polymorphism. It leads to un-decidability in program based testing as it can lead to messages sent to wrong object [33].

Polymorphic names make difficulties because they lead to un-decidability in program-based testing. Polymorphism affects on correctness of a code and cause trouble to testing.

3.3.1. Un-decidability of dynamic binding: Polymorphic names denote object of different classes so it bring difficulty in invoking operations of polymorphic names until runtime (dynamic binding) i.e. whether the original or a redefined implementation will be selected. Polymorphism brings un-decidability to program-based testing.

3.3.2. Extensibility of hierarchies: If given a polymorphic call or an operation have one or more polymorphic parameters then in test plan it found difficult to plan a test in which you check the operation with parameters of all possible classes, because classes hierarchy is freely extensible [31].

3.4. Abstract Classes

Abstract class encompasses the abstract keyword in its declaration. Abstract classes are created only to act as a base class. Abstract classes cannot be instantiated because of their full features are not completely implemented which generate challenge for executing super class. We need to test abstract test class every time because it may present errors. Derived classes from abstract classes can easily get tested [33].
### Table 1. Categorization of Automatic Test Generation Tool for Object Oriented Code

<table>
<thead>
<tr>
<th>Sr No</th>
<th>TOOL</th>
<th>INSTITUTION</th>
<th>LAST MODIFICATION ON</th>
<th>TYPE</th>
<th>METHOD</th>
<th>INPUT CODE TYPE</th>
<th>REQUIRED INPUT</th>
<th>OUTPUT</th>
<th>TESTING TECHNIQUE</th>
<th>DOMAIN</th>
<th>LICENSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test</td>
<td>Parasoft</td>
<td>2015</td>
<td>Commercial</td>
<td>Static analysis</td>
<td>JAVA</td>
<td>Source code</td>
<td>Junit test cases</td>
<td>White box testing technique</td>
<td>Desktop and server edition</td>
<td>Commercial tool with 14 days validity</td>
</tr>
<tr>
<td>2</td>
<td>C++ Test</td>
<td>Parasoft</td>
<td>2010</td>
<td>Commercial</td>
<td>Random</td>
<td>C++</td>
<td>Source and binary code</td>
<td>Unit tests</td>
<td>Static analysis</td>
<td>Desktop</td>
<td>Commercial 14 days validity</td>
</tr>
<tr>
<td>3</td>
<td>AgitarObe[48]</td>
<td>Agitar Technologies</td>
<td>2015</td>
<td>Commercial</td>
<td>Random</td>
<td>Java</td>
<td>Source code</td>
<td>Junit tests</td>
<td>Observation derived testing</td>
<td>Desktop</td>
<td>Commercial 30 days validity</td>
</tr>
<tr>
<td>4</td>
<td>CodeProAnalytix</td>
<td>Google Inc</td>
<td>2010</td>
<td>Commercial</td>
<td>Random</td>
<td>Java</td>
<td>Source code</td>
<td>Junit test cases</td>
<td>White box Testing technique</td>
<td>Desktop</td>
<td>Apache license 2.0</td>
</tr>
<tr>
<td>5</td>
<td>Randoop [15][16]</td>
<td>MIT CSAIL</td>
<td>2015</td>
<td>Open source software</td>
<td>Random</td>
<td>JAVA</td>
<td>Source code</td>
<td>Unit test suite</td>
<td>Feedback directed random testing</td>
<td>Desktop</td>
<td>MIT license</td>
</tr>
<tr>
<td>6</td>
<td>JWalk [17][18]</td>
<td>University of Sheffield</td>
<td>2013</td>
<td>Academic research</td>
<td>Random</td>
<td>JAVA</td>
<td>Source code</td>
<td>Test report</td>
<td>Easy systematic testing</td>
<td>Desktop</td>
<td>JWalk License</td>
</tr>
<tr>
<td>7</td>
<td>JCUTE [19]</td>
<td>University of Illinois</td>
<td>2006</td>
<td>Academic research</td>
<td>Symbolic execution</td>
<td>JAVA</td>
<td>Source code</td>
<td>Unit tests</td>
<td>Search based concolic testing</td>
<td>Desktop</td>
<td>free</td>
</tr>
<tr>
<td>8</td>
<td>CATG [20]</td>
<td>University of Illinois</td>
<td>2015</td>
<td>Open source software</td>
<td>Symbolic execution</td>
<td>JAVA</td>
<td>Source code</td>
<td>Test cases</td>
<td>Random testing</td>
<td>desktop</td>
<td>Open source BSD license</td>
</tr>
<tr>
<td>9</td>
<td>EvoSuite [21]</td>
<td>Research project by dr gordon Fraser and dr andrea arcuri</td>
<td>2015</td>
<td>Academic research and open source software</td>
<td>Search Based</td>
<td>JAVA</td>
<td>Source code</td>
<td>Junit tests</td>
<td>Search based approach</td>
<td>desktop</td>
<td>LGPL License</td>
</tr>
<tr>
<td>10</td>
<td>GRT [22]</td>
<td>Lastly machi,sheng,hiroyuki,johannes,rudolf</td>
<td>2015</td>
<td>Academic research</td>
<td>Random</td>
<td>JAVA</td>
<td>Source or byte code</td>
<td>Junit Test cases</td>
<td>Static and runtime analysis</td>
<td>desktop</td>
<td>Free</td>
</tr>
<tr>
<td>11</td>
<td>JTestExpert [23]</td>
<td>Abdeelrahim saiti,philips,yann-gael</td>
<td>2015</td>
<td>Academic research</td>
<td>Search Based</td>
<td>JAVA</td>
<td>Source code</td>
<td>Whole test suite</td>
<td>Search Based</td>
<td>Desktop</td>
<td>Free</td>
</tr>
<tr>
<td>12</td>
<td>Symbolic PathFinder[24]</td>
<td>Microsoft</td>
<td>2015</td>
<td>Open source software</td>
<td>Symbolic Execution</td>
<td>JAVA</td>
<td>Software</td>
<td>Test cases</td>
<td>Model checking</td>
<td>Web application</td>
<td>Used at NASA Free for research</td>
</tr>
<tr>
<td>13</td>
<td>E3 [25]</td>
<td>Yoshin puatsiya</td>
<td>2015</td>
<td>Open source software</td>
<td>Random</td>
<td>JAVA</td>
<td>Source code</td>
<td>Test suite</td>
<td>Dynamic testing</td>
<td>desktop</td>
<td>GPL (general public license)version 3</td>
</tr>
<tr>
<td>14</td>
<td>Jcrasher [26]</td>
<td>Christoph and yannis</td>
<td>2007</td>
<td>Academic research</td>
<td>Random</td>
<td>JAVA</td>
<td>Source code</td>
<td>Test File</td>
<td>Random or robust testing</td>
<td>desktop</td>
<td>Apache License</td>
</tr>
<tr>
<td>15</td>
<td>PET [27]</td>
<td>Efran and miguel</td>
<td>2011</td>
<td>Academic Research</td>
<td>Symbolic Execution</td>
<td>JAVA</td>
<td>Byte code</td>
<td>Test cases</td>
<td>White box testing</td>
<td>desktop</td>
<td>Free</td>
</tr>
</tbody>
</table>

### 4. Proposed System

Approach used following steps,

a. Input java class along with class path and class file.
b. Pre-processing is done on classes to generate instances.
c. Instantiate class under test and all other required classes.
d. Perform sequence of method call.
e. Apply genetic algorithm to reach test target.
f. Generate test data.
g. Display test cases in Junit format.

When generating the instance of java class, if it has variable, method parameter or objects which initiate with null. If it use that instance in finding the adequate sequence of method call it may generate null pointer exception. In the proposed work it avoids this null pointer exception which may terminate the execution or it takes time for code coverage. In contribution we are going to check null instances at the beginning of the
instance generation which helps us to avoid the null pointer exception later in the execution.

![Figure 2. Architecture of Proposed System](image)

Proposed system takes input as a java class path and java file to be tested and automatically produce test cases in form of Junit[34][35] System is completely automated and covers all methods in java file. Only need to provide input as (.java) file no extra information required which is easy to use even for novice user. System approach developed in two phases like a pre-processing phase and a test-data generation phase.

4.1. Pre-processing for Instance Generation

In this phase we are parsing java files and explore abstract syntax tree (AST) for it using Eclipse JDT [36]. JDT makes our static analysis easy because it allows creating an AST visitor for each essential information.in this first stage we are using static analysis of all methods and collecting all necessary information required for second step i.e., test data generation.in this first stage we are giving java files as input and according to that Abstract syntax tree (AST) [37] is generated.

An AST is a tree model of an whole program e.g., Java program a statement or an expression.

Example-

While (k < 7) {
    foo (k);
    K++;
}
In this phase AST of java file which is under test modified to call specific method on entering new branch after covering all methods java file get saved and compiled to generate (.class) file i.e., java byte code. System performs pre-processing in following steps like, Method Instrumentor and Analyzers.

4.1.1. Method Instrumentor: To instrument the Java file under test, its AST is modified to call a specific method on entering each branch. This method call takes as inputs the branch code and notifies the testing process when the branch is executed. After the instrumentation, based on the AST, a new version of the Java file under test is saved and compiled to generate its new Java byte code file (.class).after that.

4.1.2. Analyzer: To extract the information required for the problem representation and the instance generator, several assessments of the AST are performed:

a. Identify all branches wherein the particular method is called;
b. identify all branch modifiers for each data member;
c. Constant values are saved for strings and each primitive type

To simplify the implementation of parsing the Java file under test and exploring the AST, we used the parser provided with Eclipse JDT [36]. JDT makes our static analysis easy because it allows creating an AST visitor node for each requires information.

4.2. Test Data Generation

The test-data generation phase is main aim of system which generate test cases for all methods present in java file and satisfy test coverage.

This phase include following steps like,

a. Generation of object instances.
b. Call sequence of method.
c. Applying genetic algorithm.
d. Test data generation.
e. Display Junit test cases.
4.2.1. Generation of instances: In object oriented code to call method or constructor we need some instances of classes. Instance generation is main task because sufficient instances needed to cover all branches. Instance generation plays important role in test data generation instance indicates the relationship of an object to its class. It provide way and extra information for further steps like sequence of method call.

In this phase we are considering 3 types of classes
a. Atomic class – all string classes as well as primitive types, classes which encapsulate only primitive types are considered Atomic.
   b. Container class- A Container is grouping objects like, List and Array, i.e., it is an object that can encompass other objects, often referred to as its elements. For container class random instance generator is used bound length then it recursively call which randomly select and for other class means-of-instantiation is used.
   c. Simple classes or other classes than container or atomic classes.

In this phase we generate instances of class.

Algorithm 1. Generation of instances
Input- java program file having class is to be instantiated
Output- instance of class
Step 1. identify class type
Step 2. if class is atomic or container generate instance by random instance generation
Step 3. for other classes by using JAVA Reflection API generate means of instantiation.
Step 4. calculate complexity of selected means of instantiation.

In this phase input is java class file to generate all needed elements then consider class type and according to that generated and a mean-of-instantiation is selected. To generate this set, for a given input class, we use the Java Reflection API [38] to get the means-of-instantiations offered by that class and the open-source library Reflections [39] to get subclasses because subclasses decide which class to instantiate and external factory methods.

4.2.2. Generator of Sequences of Method Calls: Sequence of method calls are required to put the input class in a desirable state to reach a test target. A sequence of method calls on that instance. The sequence of method calls can be split into two steps,
   a. Putting the input java class instance in an adequate state;
   b. Achieving the test target.

Algorithm 2. Generator of Sequences of Method Calls
Input - methods.
Output- test data candidate.
Step 1. instances for class generated.
Step 2. generate sequence of state modifier.
Step 3. select method randomly from sequence of state modifier.
Step 4. instances of class generated in loop.
Step 5. target method is generated.

4.2.3. Genetic Algorithm: GA has steps like population, selection, crossover, and mutation that has been applied in many areas [40]. Adoptive search techniques are does not find the optimal solution at all time, however they often find a very good solution in limit of time [45]. GA is used to produce test data because their robustness for solutions of different test tasks. [41], [42], [43.], [44] Genetic algorithms provide guarantee high probability of enhancing the quality of each individuals over several generations according to the Schema Theorem [45].

Simple Genetic Algorithm It works as follows:
a. Randomly generate a population of n individuals;
b. Evaluate the fitness f(i) of each individual i in the population;
c. Exit if a stopping condition is fulfilled;
d. Repeat the following steps until n number of offspring have been created:
   a. From the current population select a pair of parents;
   b. Create two new offspring and Recombine the two selected parents and;
   c. Apply mutation on two offspring.
e. Replace the population by new population and return to Step 2.

As an example, a simple genetic algorithm is given below:
{
    Initialize population;
    Evaluation of population;
    While Criteria of Termination Not Satisfied
    {
        Select parents for reproduction;
        Perform recombination and mutation;
        Population evaluation;
    }
}

4.2.4. Test Data Generator: Test-data generation is one of the most costly parts of the software testing phase. Therefore, automating this task can significantly reduce software cost. Algorithm presents the different steps of this component to satisfy the all target branch coverage criterion for a java file under test and generate test cases in Junit format. It use JDT which translate the set of test-data into a Java file that contains test cases in Junit format.

Algorithm 3. generation of test data
Input - java file
Output - test data
Step 1. Analyze java file which is given as input to generate test data.
Step 2. To cover each branch select domain vector.
Step 4. Cover some uncovered branches.
Step 5. Translate set of test data into java file that contain test cases in Junit format.

4.2.5. Junit test cases: For the Java Programming Language Junit is a unit testing framework. Junit work based on setting up the test data for a portion of code which can be first tested and then can be implemented. This approach improves the productivity of programmer as well as stability of program code that decreases programmer stress and the time spent on debugging.

Features of Junit
   a. Junit is used for writing & running tests which is an open source framework.
   b. For testing expected results Provides Assertions.
   c. Test runners Provides for running tests.
   d. Junit tests allow writing code faster which improves quality.
   e. Junit is simple, less complex and takes less time.

5. Implementation and Results
We have implemented our approach as a web based application, that takes input java files and class path and produce test cases in Junit format.
5.1. Input to Proposed System

For proposed system we are providing java classes along with class path is shown in Figure 4.

![Figure 4. Input to Proposed System](image)

5.2. Selection of target method

To reach test target or to cover all branches we are providing test target is shown in Figure 5.

![Figure 5. Selection of Target](image)
5.3. Output

System automatically generates Junit test cases is shown in figure 6.

![Figure 6. Output](image)

5.4. Experimental Subject of input

Here we are providing 2 input libraries table 2. Provide details of it.

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Classes</th>
<th>Method</th>
<th>Branch</th>
<th>Lines</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input1</td>
<td>5</td>
<td>19</td>
<td>5</td>
<td>171</td>
<td>10</td>
</tr>
<tr>
<td>Input2</td>
<td>6</td>
<td>27</td>
<td>12</td>
<td>253</td>
<td>26</td>
</tr>
<tr>
<td>All</td>
<td>11</td>
<td>46</td>
<td>17</td>
<td>424</td>
<td>36</td>
</tr>
</tbody>
</table>

5.5. Comparison of other approach with genetic algorithm

Following graph shows code coverage with other approaches with genetic algorithm when merged to it is shown in Figure 7.

![Figure 7. Comparison of Other Approach with Genetic Algorithm](image)
6. Conclusions

The features of GAs make the test case generation process easy also overcome the drawback of random searching. Proposed system developed to generate Junit test cases automatically which helpful to all developers, here approach of existing system combined with Genetic algorithm to improve the code coverage.

References

[10] Voas , Morell and Miller, "Predicting where faults can hide from testing", IEEE vol. 8, pp. 41-48.


[34] Junit is a simple framework to write repeatable tests [Online], (2013). Available: http://www.junit.org


[36] E. J. development tools (JDT), The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins [Online],(2013), Available: http://www.eclipse.org/jdt


[38] The java reflection API [Online], (2013), Available: http://docs.oracle.com/javase/tutorial/reflect


