International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-8 Issue-10, August 2019

Automatic JUnit Generation and Quality Assessment using Concolic and Mutation Testing

Avadhani Bindu, Saumya Ranjan Giri, P. Venkateswara Rao

Abstract*: The necessity of Unit Testing cannot be denied in catching bugs at the lowest level of software development with low cost of fixing bugs. It is difficult and costly to detect a bug at a later stage of development in a large module of software. Since there are many individual units (functions) in a software program, manual unit testing needs a lot of effort and time. A huge amount of time and human effort can be saved if unit testing is automated. There are 3 basic needs of unit testing i.e. identifying the inputs to a function, detecting runtime errors and detecting logical errors. Concolic Testing is used to identify the inputs to function and detecting the runtime errors. Mutation Testing is used to detect the logical errors. The identified inputs to a function are called as Test Cases. Mutation testing can verify the test cases to see if there is a need of having more test cases for a function. Our software uses both Concolic and Mutation testing and can automate the unit testing process of Java code to a great extent. The output of the work is JUnit where user can make his own assertions for every auto-generated test cases.

***Keywords*: Unit Testing, JUnit, Concolic testing, Mutation testing, Java Path Finder and PIT.**

I. INTRODUCTION

A bug-free system is the main reason to implement the software testing. A little bug can create an immense problem where it is the scope for the hackers to steal the private data which is an intolerable act. The testing helps us to check the system's functionality correctness. Hence, software testing plays an important and very crucial role in detecting the errors in the complex systems.

Among the types of testing techniques, the unit testing is the technique performs at code level in finding bugs by isolating the piece of code which is called as Unit. In order to make the system cost-effective unit testing is the earlier stages of testing techniques performed for early detection of bugs.

II. CONCOLIC TESTING:

Concolic testing is a hybrid software testing technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables along a concrete execution (testing on specific inputs) path.

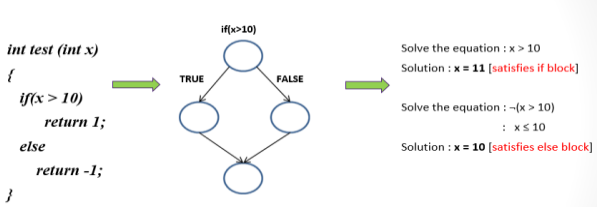

Fig 1: Concolic Execution

The method test (int x) in the above Fig 1 has two paths, which forms the execution tree. These paths can be executed by running the program on the inputs x=11 and x=10. The test suite generation is achieved with the help of the constraint solvers like Z3, CORAL, SMTInterpol by checking the condition x > 10 to cover the true path and by negating the condition x > 10, which is x<= 10 to cover the false path.

MUTATION TESTING:

The mutation testing technique was given birth by Richard Lipton with his proposal in 1971 and was developed. It was published in late 1970s by DeMillo et.al and Hamlet [19, 20]. First implementation of mutation testing tools was done by Timothy Budd as a part of his Ph.D. work from Yale University in 1980[18]. Adequately measuring the quality of testing is hard and the researchers have proposed some metrics lying on the notion of code coverage, which is the description of how much source code is covered by the test suites. Code coverage metrics plays a vital role in critical projects like auto-pilot system, car air bag system etc., it checks if all instructions are tested and how well it is tested. Mutation testing is the effective technique where the quality assessment of test suites is done. It gives us better understanding of what tests exercise on the program. Upon introducing faults into the code the test cases quality is being checked where the output value differs from the original code to mutated code. The overview of mutation testing is picturized as shown below in Fig 2.

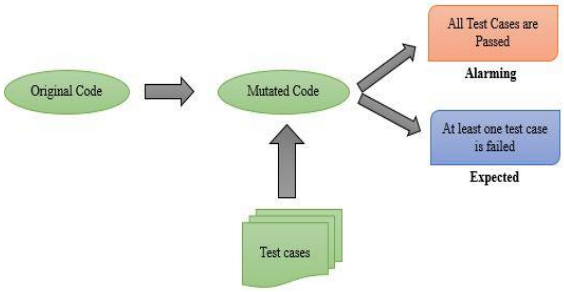

Fig 2: Mutation Testing

The goal of this technique is to assess the quality of the test suites which should be robust to fail the mutated code. It is also known as fault-based testing strategy as it is involved in introducing the faults into code.

Manuscript published on 30 August 2019.
*Correspondence Author(s)
Avadhani Bindu*, Department of Computer Science and Engineering, VNR Vignana Jyothi Institute of Engineering and Technology, Hyderabad, India.
Saumya Ranjan Giri, Robert Bosch Engineering and Business Solutions Pvt. Ltd., Bengaluru, India.
P. Venkateswara Rao, Department of Computer Science and Engineering, VNR Vignana Jyothi Institute of Engineering and Technology, Hyderabad, India.

If P (original program) is a program and upon injecting the faults to the program P it is said to be mutated i.e. P′ (mutated program). In the process of mutant generation the faults are introduced into the original program by replacing the operators of the original program where that operator is called mutant operator. The outcome of the mutation testing is the mutation score which is the ratio of number of killed mutants to the total number of mutants generated.

Mutation Score = Killed Mutants/ Total no.of Mutants * 100.

The test set when executed using mutation testing, if the result of mutated program differs from the result of original program is said to be a mutant *killed*. If the result of mutated program do not differs from the result of original program is said to be a mutant *survived*. If the behavior of the resultant of mutant is exactly same as the original they are called equivalent mutants which are logically same.

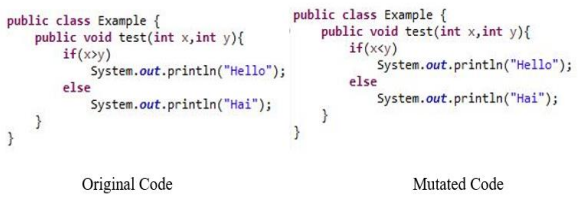

Fig 3: Example of Mutation

In the above example from Fig 3 the conditional operator "if(x > y)" in original code will be mutated to "if(x < y)" which is a mutated code.

III. BACKGROUND OF THE WORK

Cristian Cadar and Koushik Sen [1] described that Concolic testing [14] performs symbolic execution while the program is executed using concrete values. It is said that it generates high coverage test inputs. He also described that the symbolic execution generates the set of input values by exercising the path and checks for the errors including uncaught exceptions and run-time errors, memory corruption. Symbolic execution represents the program variables as the symbolic variables over the symbolic input values. DART, CUTE and CREST are the tools for the concolic testing where DART is the first concolic testing tool and jCUTE is for java after the CUTE [13, 14].

[2] Java Path Finder, the explicit state model checker for Java programs built on the top of a virtual machine. It is able to handle all the language features of Java. It is used for deep inspection (numerical analysis), software model checking and test case generation (symbolic execution).

Corina. S. Pasareanu [3] Symbolic execution is enabled by JPF-SE helps in checking the behavior of the code with usage of symbolic values, automates the test input generation for java library classes and is extended to handle the decision procedures. JPF-SE, the extension of JPF [2]Using different decision procedures JPF is used to check the programs and generate the test inputs.The numeric values, complex data structures are handled by its instrumentation package.

Corina S. Pasareanu, W.Visser et.al [4] tells us that Symbolic execution is for efficient test input generation[5] but it is focused mostly on generating test inputs for simple data types(integers for the most part).

Corina S. Pasareanu et.al [5] developed the framework on symbolic execution that handles the dynamically allocated structures (lists etc.) and for primitives data and concurrency.

Kasper Luckow et.al [6] tells us that JDART, a dynamically symbolic analysis framework for Java is extended with a component that efficiently constructs constraints with constraint solvers. CORAL, SMTInterpol and Z3 solvers are supported by it and is able to handle the NASA software with constraints containing bit operations.

The Dynamic Symbolic execution executes the program with concrete and symbolic inputs at the same time. It maintains the path constraints whenever the branching condition in the code is encountered. Combined execution paths forms the constraints tree which heuristically exercises these paths with a constraint solver. Constraint solvers helps in solving the complex constraints using specialized solvers.

Corina S. Pasareanu et.al [7] tells us that symbolic execution is a powerful technique for the high coverage automated test inputs generation. It is shown that this is powerful than DART where it fails to handle the classical symbolic execution techniques due to incompleteness of decision procedures.

Corina.S. Pasareanu et.al [8] this work implements a non-standard interpreter on the top of JPF. It is to improve the precision of the symbolic analysis. It internally tells us that Concolic execution [15, 16] is an analysis technique that performs a concrete execution, random test cases and it collects the path constraints along the execution path which are solved with the help of constraint solvers.

Corina. S. Pasareanu and Willem Visser et.al [9] is the description of the JPF-core and its extension to the symbolic execution JPF-symbc internal architecture. It describes JPF-symbc that combines symbolic execution with model checking and constraint solving for automated error detection and test input generation for Java Programs. JPF is open sourced in 2005.

Henry Coles et.al [10] is the description of the PIT the mutation testing tool by presenting that PIT is a mutation testing tool for Java which is robust, scalable, and fast and is well integrated with Ant and Maven and is an open source which is actively maintained.

Forostyanova Mariya et.al [11] with its comparison between µJava and Pitest it tells us that Pitest generates less number of mutants and helps in avoiding the generation of equivalent mutators.

Zainab Nayyar et.al [12] paper tells us that Mutation testing is helpful for verification and validation of the requirements. It not only ensures the good quality of test cases but also the faults in the programs.

IV. EXISTING METHOD

i) Test case Generation

Java Path Finder:

JPF core is a Virtual Machine (VM) for Java byte code. It is a program that executes the system under test (SUT). The VM of the JPF handles the byte code instructions created by the Java Compiler. It itself is written in java. It is the framework which runs on the top of JVM. Some of the typical uses of JPF are the deep inspection (numerical analysis), the test case generation (Symbolic Execution). [2, 17]. [9] Tells us that JPF has been under development by NASA Ames Research Center since 1999 and open sourced in the year 2005. JPF is now available in the github repositories which can be built using Apache Ant.

To install JPF-core and JPF-symbc, to create site.properties file, to run the JPF using command line, creating .jpf configuration file, the complete documentation is provided and can be followed accordingly [2,17]. Let us consider the following example in Fig 4 which handles only primitives.

```
public class ExampleProgram {
    int test (int x){
       if(x > 10)
            return 1;
       else
           return -1;
    }
```

Fig 4: Example for method with primitive as argument

For the given example the JPF by executing the command of JPF [17] it gives the test cases as follows in fig 5:

```
========================================== Method Sequences
[test(11)]
[test(0)]
```

Fig 5: Method sequences by JPF for primitives

Let us consider the example in Fig 6 which handles the objects.

```
public class Department {
    private int deptId;
    private String deptName;
    private List<Student> studentList;

    public int getDeptId() {
        return deptId;
    }
    public void setDeptId(int deptId) {
        this.deptId = deptId;
    }
    public String getDeptName() {
        return deptName;
    }
    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }
    public List<Student> getStudentList() {
        return studentList;
    }
    public void setStudentList(List<Student> studentList) {
        this.studentList = studentList;
    }

}
```

An example of Department Model class

```
public class Student {
    private String Sname;
    private int rollNo;
    public String getSname() {
        return Sname;
    }
    public void setSname(String sname) {
        Sname = sname;
    }
    public int getRollNo() {
        return rollNo;
    }
    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}
```

An Example of Student Model Class

```
public class SUT {
    public static double testMe(Department d,int x){
        if(x > 100 && d.getDeptId() == 80 && d.getDeptName().equals("CSE")
                && d.getStudentList().get(0).getSname().equals("XYZ")){
            x++;
            return 50.0;
        }else
            return 40.0;
    }
```

An Example of a method with Object as argument

Fig 6: Example of Object

For the above example in the Fig 6 for objects the JPF by executing the command of JPF [17], the results are as follows:

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

```
====================================================== Method Sequences
[(expected = java.lang.NullPointerException.class), testMe(-9223372036854775808,101), ##EXCEPTION## "java.lang.NullPoint
erException: Calling 'getDeptId()I' on null object..."]
[testMe(-9223372036854775808,101)]
[(expected = java.lang.NullPointerException.class), testMe(-9223372036854775808,101), ##EXCEPTION## "java.lang.NullPoint
erException: Calling 'equals(Ljava/lang/Object;)Z' on null object..."]
[testMe(-9223372036854775808,0)]
```

Fig 7: Method Sequences by JPF for objects

ii) Test case Verification:

The comparative study of the mutation testing tools for java along with experimentation we found that PIT is fast, scalable and the has less mutant generations which also avoids generating equivalent mutants[18]. The work of Henry et.al [10] helps us to know that PIT generates mutants using byte code manipulation and keeps the memory overhead as low. The mutated code is generated by the scanning process but it is discarded. To install PIT and to run using the command line, utilizing its options are described by Henry Coles. The report can be viewed in different formats like XML, HTML, and CSV in PIT. It also has set of mutators which are by default active and inactive [18].

V. PROPOSED METHOD

i). Test Case Generation for derived data types and complex data structures using JPF and Junit generation:

As mentioned in the related work section that JPF-core is the framework which has its extensions like JPF- symbc which together helps us in deep inspection and test case generation, the proposed work utilizes the JPF to generate the test cases. Upon experimentation result from the above example shown in the Fig 6 it is seen that test cases are not generated for object type i.e. Department as shown in the Fig 7, as it has done for the primitives by evaluating the condition. So, it can be said that JPF can generate the test inputs for the primitive and String data. Hence this work is proposed to generate the test inputs for the objects (derived data types) and for complex structures (like list containing list etc).

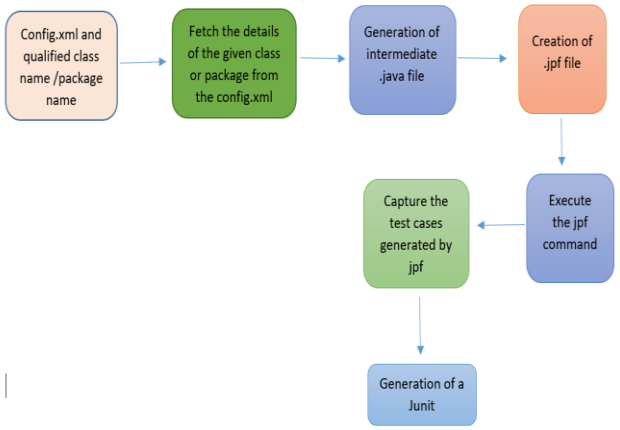

Fig 8: The Process of Test case Generation

The Config.xml file is created such that it has the target class sourcepath, classpath, Junit source path to store the created Junit. With the help of given qualified class name and the Config.xml file contents, the details of the System Under Test or Target (class name /package name) are fetched. Then for the Target/SUT, the intermediate java file is created such that if there exists the objects or complexed data structures like list of lists, list of array of objects etc., as the method parameters the objects are broken down into primitive level. Let us consider an example as shown below in Fig 6 where Department, Student are the model classes with getter and setter methods and the example of Object as the argument in the class SUT with method testMe (Department d, int x) where one parameter is the object and other is the primitive. The intermediate java file generated by our system for the method *testMe (Department d, int x)* in the Class SUT is as shown below in Fig 9.

```
public class IntermediateSUT {
    public static void wrapperMethod(int arg0_0,String arg0_1,String arg1_0,int arg1_1,String arg0_2_0_0,
                                     int arg0_2_0_1,String arg0_2_1_0,int arg0_2_1_1,int arg2){
        Department department = new Department();
        department.setDeptId(arg0_0);
        department.setDeptName(arg0_1);
        Student student = new Student();
        Student student_1 = new Student();
        student.setSname(arg0_2_0_0);
        student.setRollNo(arg0_2_0_1);
        student_1.setSname(arg0_2_1_0);
        student_1.setRollNo(arg0_2_1_1);
        List<Student> sList = new ArrayList<>();
        sList.add(student);
        sList.add(student_1);
        department.setStudentList(sList);
        SUT.testMe(department, arg2);
    }

    public static void main(String[] args){
        wrapperMethod(50, "ABC", " ", 0, "XYZ", 20, "NJU", 30, 200);
    }
}
```

Fig 9: Intermediate Java file

The intermediate java file with the name of Intermediate followed with the class name with a wrapperMethod with parameters as primitives of Department object and the last primitive parameter. Here the example has Department object has broken down as shown below in the Fig 10

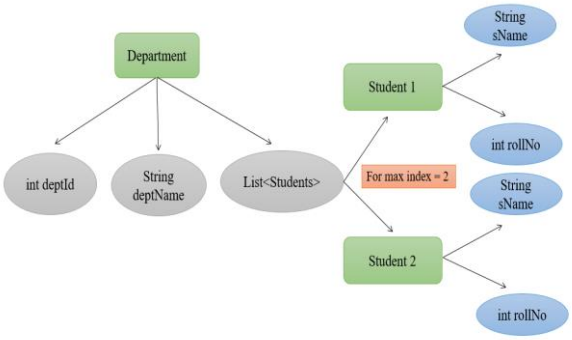

Fig 10: Breakage of Objects or complexed structures into primitives.

Object for the class Department is created and the primitives of the Department class deptId, deptName, List of Students containing Student name and rollNo are assigned with variable names declared in wrapperMethod parameters and are set to the respective members of the objects. Finally the target method is invoked. Here the max index value is 2 that means the List creation will be considered till the index 2. Likewise it is also applicable to arrays, sets, and maps. Since JPF works for only fixed size data, we need this max index value to expand array, set, list, and map.

Now the target method to the JPF is the intermediate java file. After the test inputs are generated by the JPF (for possible cases) they are wrapped into objects and a systematic and syntactically correct Junit is generated.

JUnit Generation for Protected and Private Methods Using Java Reflection:

Due to inaccessibility of private and protected methods outside the class it is difficult to generate the test suites for those methods, Java Reflection API helps us to invoke the private/protected methods which makes the testing of private/protected methods is made easy.

Reflection API is used to examine or modify the behavior of methods, classes, interfaces at run-time. Java.lang.reflect package provides us the required classes for reflection. Through the reflection methods can be invoked at run-time irrespective of the access specifier used with them.We can invoke a method through reflection if we know its name and parameter types. We use below two methods for this purpose.

1. .getDeclaredMethod (): To get the declared methods of a class.

invoke (object, parameter): To invoke a method of the class at run-time.

If the method of the class doesn't accepts any parameter the null is passed as argument.

2. Through reflection we can access the private variables and methods of a class object and invoke the method by using the object. We use below two methods for this purpose.

- Class.getDeclaredField (Fieldname): Used to get the private field. Returns an object of type field for specified field name. Here the field is the method name.
- Field.setAccessible (true): Allows us to access the field irrespective of the access modifier used in the field.

ii). Test case Verification and generation of additional TC:

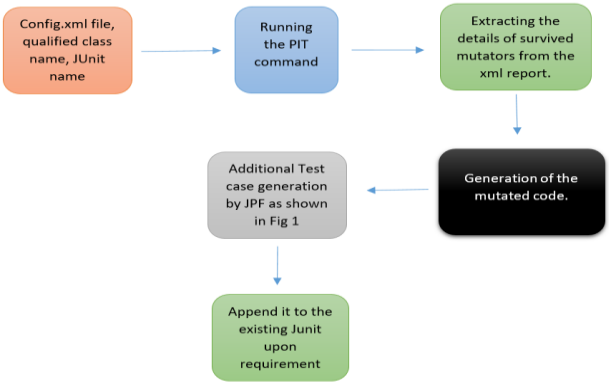

Fig 11: The Process of Test case Verification

As we know that PIT does the byte code manipulation and it can generate the report in HTML, XML formats and it can give the report in both formats for the single run [18]. Upon deep analyzation and experimentation of PIT and its procedure from [18] we have proposed the idea of generating the mutated code from the copy of original code by utilizing the XML report generated by the PIT. The XML report of PIT has the following contents as shown in Fig 12 which is the sample XML report generated by PIT.

```
<?xml version="1.0" encoding="UTF-8"?>
<mutations>
  <mutation numberOfTestsRun="2" status="SURVIVED" detected="false">
    <sourceFile>Mutation.java</sourceFile>
    <mutatedClass>com. bindu. Mutation</mutatedClass>
    <mutatedMethod>test</mutatedMethod>
    <methodDescription>(Lcom/bindu /Person;)I</methodDescription>
    <lineNumber>43</lineNumber>
    <mutator>org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator</mutator>
    <index>33</index>
    <block>5</block>
    <killingTest/>
    <description>changed conditional boundary</description>
  </mutation>
</mutations>
```

Fig 12: Sample XML report generated by PIT

As it is already described in the documentation of PIT[18] that what operators are mutated into the other operator w.r.t the mutator name, from the XML file as shown above the line number, mutator, mutated method, status, source file, mutated class are utilized to generate the mutated code.
From the Fig 11 we can say that first with help of the given input i.e. class name and method name and the XML report of the PIT the source file to be mutated is fetched and copied if the mutations generated are survived. Then with the help of line number and mutator name, code is mutated only at that line

number and mutator name, code is mutated only at that line number with the given mutator w.r.t the mutation operators (eg: > to >= if it is Conditionals_Boundary mutator) [18] and is given to the concolic engine to generate the set of test cases. Once the test cases are generated, they are captured and the mutated code is deleted. Upon comparing the test cases generated with the existing JUnit, if there exists any new test cases other than the existing test cases they are appended to the JUnit, which are the additional test cases generated by the Concolic engine to kill the survived mutations. When the Concolic engine is not able to generate the additional test suites user has to evaluate and add them manually.

VI. OUTCOMES OF THE WORK

For Primitives:
For the example shown in Fig 4 the JUnit generated is as follows in the Fig 13.

```
public class ExampleProgramTest {

    // Launch the test
    public static void main(String[] args) {
        new org.junit.runner.JUnitCore().run(ExampleProgramTest.class);
    }


    @Test
    public void testTest_1() throws Throwable {
        ExampleProgram exampleProgram_obj = new ExampleProgram();
        int ret_test = exampleProgram_obj.test(11);
        //TODO assert here
        //int expectedOutput = 0;
        //assertEquals(expectedOutput, ret_test);
    }


    @Test
    public void testTest_2() throws Throwable {
        ExampleProgram exampleProgram_obj = new ExampleProgram();
        int ret_test = exampleProgram_obj.test(0);
        //TODO assert here
        //int expectedOutput = 0;
        //assertEquals(expectedOutput, ret_test);
    }

}
```

Fig 13: JUnit for the example in Fig 4

Then upon verifying test cases the following are the test case is generated to kill the survived mutations is shown in the Fig 14.

```
@Test
public void testTest_4() throws Throwable {
    ExampleProgram exampleProgram_obj = new ExampleProgram();
    int ret_test = exampleProgram_obj.test(10);
    //TODO assert here
    //int expectedOutput = 0;
    //assertEquals(expectedOutput, ret_test);
}
```

Fig 14: Additional Test Case generated for the example in Fig 4

For Objects:
Total 5 test cases are generated for the method testMe (Department d, int x) in SUT class shown in Fig 6 by JPF. A portion of the JUnit generated for the example shown in the Fig 6 is as shown below.

```
@Test
public void testTestMe_1() throws Throwable {
    Department department_0 = new Department();
    department_0.setDeptId(0);
    department_0.setDeptName("CSE");
    Student student_0 = new Student();
    student_0.setSname("XYZ");
    java.util.List<com.bindu.Student> studentList_0 = new java.util.ArrayList<>();
    studentList_0.add(student_0);
    department_0.setStudentList(studentList_0);
    double ret_testMe = SUT.testMe(department_0, 101);
    //TODO assert here
    double expectedOutput = 40.0;
    assertEquals(expectedOutput, ret_testMe, 0.001);
}

@Test
public void testTestMe_2() throws Throwable {
    Department department_0 = new Department();
    department_0.setDeptId(0);
    department_0.setDeptName("CSE");
    Student student_0 = new Student();
    student_0.setSname("XYZ");
    java.util.List<com.bindu.Student> studentList_0 = new java.util.ArrayList<>();
    studentList_0.add(student_0);
    department_0.setStudentList(studentList_0);
    double ret_testMe = SUT.testMe(department_0, 0);
    //TODO assert here
    double expectedOutput = 40.0;
    assertEquals(expectedOutput, ret_testMe, 0.001);
}
```

Fig 15: Test Cases generated for Object's example in Fig 6

Upon verifying the test cases, following additionally generated test cases are added (shown in Fig 16) to kill the survivals of the example shown in the Fig 6.

```
@Test
public void testTestMe_3() throws Throwable {
    Department department_0 = new Department();
    department_0.setDeptId(80);
    department_0.setDeptName("CSE");
    Student student_0 = new Student();
    student_0.setSname("XYZ");
    java.util.List<com.bindu.Student> studentList_0 = new java.util.ArrayList<>();
    studentList_0.add(student_0);
    department_0.setStudentList(studentList_0);
    double ret_testMe = SUT.testMe(department_0, 100);
    //TODO assert here
    //double expectedOutput = 0.0;
    //assertEquals(expectedOutput, ret_testMe, 0.001);
}


@Test
public void testTestMe_4() throws Throwable {
    Department department_0 = new Department();
    department_0.setDeptId(0);
    department_0.setDeptName("CSE");
    Student student_0 = new Student();
    student_0.setSname("XYZ");
    java.util.List<com.bindu.Student> studentList_0 = new java.util.ArrayList<>();
    studentList_0.add(student_0);
    department_0.setStudentList(studentList_0);
    double ret_testMe = SUT.testMe(department_0, 100);
    //TODO assert here
    //double expectedOutput = 0.0;
    //assertEquals(expectedOutput, ret_testMe, 0.001);
}
```

Fig 16: Additional Test cases for Object's example in Fig 6

For the private method/protected method the JUnit generation is as follows:

```
private String testMe(int x, int y){
    if(x > 0)
        return "This is a positive number";
    else if(x < 0)
        return "This is a negative number";
    else
        return "This is zero";

}
```

Fig 17: Example of Private Method

The JUnit generated for the example of private method shown in Fig 17 is as shown below in Fig 18.

```
public class TestPrivateMethodTest {

    /* Private Method Test */

    private static TestPrivateMethod com_vnr_bindu_examples_testPrivateMethod;
    // Launch the test
    public static void main(String[] args) {
        new org.junit.runner.JUnitCore().run(TestPrivateMethodTest.class);
    }
    @BeforeClass
    public static void setUp() throws Exception {
        com_vnr_bindu_examples_testPrivateMethod = new TestPrivateMethod();
        // write set up code here
    }

    @Test
    public void testTestMe_1() throws Throwable {
        Method[] methods = com_vnr_bindu_examples_testPrivateMethod.getClass().getDeclaredMethods();
        for(Method method : methods) {
            if(method.getName().equals("testMe") && Modifier.isPrivate(method.getModifiers())) {
                Parameter[] params = method.getParameters();
                if(params.length == 2 &&
                        params[0].getParameterizedType().toString().equals("int") &&
                        params[1].getParameterizedType().toString().equals("int")) {
                    method.setAccessible(true);
                    String ret_testMe = (String)method.invoke(com_vnr_bindu_examples_testPrivateMethod, 0, 0);
                    //TODO assert here
                    assertNotNull(ret_testMe);
                    break;
                }
            }
        }
    }
}
```

Fig 18: JUnit for Private Method

JPF is also able to catch the run time exceptions. Our system can pick the exceptions notified by JPF and place it in proper place inside the JUnit.

```
public int myMethod(int x, int y) {
    int z =(2 * y) - 10;
    if(x == 1000){
        return x/z;
    }
    return 0;
}
}
```

Fig 19: An Example for Run-time Exception

Fig 19 is an example which has a scope of having run-time exception in the class Example. The JUnit generated and the exception divide by zero caught in the example shown in the Fig 20 is showing the user in the form of comment above that @Test method causing exception as shown below in the Fig 20.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

```
public class ExampleTest {

    // Launch the test
    public static void main(String[] args) {
        new org.junit.runner.JUnitCore().run(ExampleTest.class);
    }

    @Test
    public void testMyMethod_1() throws Throwable {
        Example example_obj = new Example();
        int ret_myMethod = example_obj.myMethod(1000, 0);
        //TODO assert here
        //int expectedOutput = 0;
        //assertEquals(expectedOutput, ret_myMethod);
    }

    @Test//(expected = java.lang.ArithmeticException.class)
    public void testMyMethod_2() throws Throwable {
        Example example_obj = new Example();
        int ret_myMethod = example_obj.myMethod(1000, 5);
        //TODO assert here
        //int expectedOutput = 0;
        //assertEquals(expectedOutput, ret_myMethod);
    }

    @Test
    public void testMyMethod_3() throws Throwable {
        Example example_obj = new Example();
        int ret_myMethod = example_obj.myMethod(0, 5);
        //TODO assert here
        //int expectedOutput = 0;
        //assertEquals(expectedOutput, ret_myMethod);
    }
}
```

Fig 20: Junit generated with caught run-time exception

VII. DISCUSSION AND EXPERIMENTAL EVALUATION

The work is experimented on the example code as well as the real code. The experimental results tell us that it generates the Junits in less amount of time.

Table-I: Experimental evaluation

S.No	Real Project / Example Project	Package Name	Number of Classes	Number of Methods	Total time taken	Total no. of @Test methods
1	Example Project	Package 1	94	143	300 seconds	558
2	Real Project-1	Package 2	11	37	7 seconds	53
3	Real Project-2	Package 3	10	99	180 seconds	148
4	Real Project-3	Package 4	6	75	120 seconds	102
5	Real Project-4	Package 5	13	44	60 seconds	81
6	Real Project-5	Package 6	15	61	180 seconds	118
7	Real Project-6	Package 7	9	88	60 seconds	135
8	Real Project-7	Package 8	10	65	45 seconds	125
9	Real Project-8	Package 9	12	60	240 seconds	108
10	Real Project-9	Package 10	10	78	180 seconds	115

Note: *To comply with information security, the actual name of the real projects and packages are not given in the above table.

From the above experimental results it can be said, if the JPF is trying to evaluate the conditions encountered in the code (If available) it generates the test cases more in number by satisfying the every possible path. Though it is a real project sometimes JPF may not generate the test cases. So, this work will help us to generate the template @Test methods of the Junit which would take very less time when JPF is not able to generate the test cases.

VIII. CONCLUSION AND FUTURE WORK

This work is a combination of

i) Automated Test Suite and JUnit generation and
ii) Verification of the test suites by generating the additional test suites.

We have presented the process of the automated JUnit generation using the Concolic testing. It internally tries to evaluate the conditions present in the branching statements of the code.

It compiles the generated intermediate java file in order to overcome the limitations of the JPF (not generating for objects as shown in the Fig 8). This work is also able to generate the JUnit with random test cases if the JPF fails to generate them. We have generated the mutated code with the help of PIT produced xml files and the same mutated code is fed to our Concolic engine to get additional test cases. If the return values are same for different branches within the same method the PIT generated mutants become equivalent.
Presently JPF is not able to evaluate Map and Set. As it is an open source, one can find the solution to make it work for Map and Set. As the Method handlers are faster than reflection API, they can be used instead of reflection API for private/protected methods invocation inside JUnit.

ACKNOWLEDGEMENT

We are immensely grateful and express our gratitude to **Dr. Corina. S.Pasareanu,** Researcher, NASA Ames Research Center, for her valuable work, suggestions, and help provided while working during this course of research.

REFERENCES

1. Symbolic Execution for Software Testing: Three Decades Later by Cristian Cadar and Koushik Sen. In Communications of the ACM, February 2013, Vol.56 No.2
2. Java Path Finder. http://javapathfinder.sourceforge.net.
3. S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In TACAS'0, 2007.
4. Willem Visser, Corina S. Pasareanu and Sarfaz Khurshid. Test Input Generation with Java PathFinder by In ISSTA'04, July 2004.
5. Khurshid S., Pasareanu C.S., Visser W. (2003) Generalized Symbolic Execution for Model Checking and Testing. In: Garavel H., Hatcliff J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2003. Lecture Notes in Computer Science, vol 2619. Springer, Berlin, Heidelberg. ISBN: 978-3-540-00898-9 , February 2003.
6. Luckow, Kasper Søe, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric and Vishwanath Raman. "JDart: A Dynamic Symbolic Analysis Framework." TACAS (2016).
7. Pasareanu, Corina S., Neha S. Rungta and Willem Visser. "Symbolic execution with mixed concrete-symbolic solving." ISSTA (2011).
8. Pasareanu, Corina S., Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person and Mark Pape. "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software." ISSTA (2008).
9. Pasareanu, Corina S., Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz and Neha S. Rungta. "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis." Automated Software Engineering 20 (2013): 391-425.
10. Coles, Henry & Laurent, Thomas & Henard, Christopher & Papadakis, Mike & Ventresque, Anthony. (2016). PIT: a practical mutation testing tool for Java (demo). 449-452. 10.1145/2931037.2948707.
11. F. Mariya and D. Barkhas, "A comparative analysis of mutation testing tools for Java," *2016 IEEE East-West Design & Test Symposium (EWDTS)*, Yerevan, 2016, pp. 1-3.
12. Z. Nayyar, N. Rafique, N. Hashmi, N. Rashid and S. Awan, "Analyzing test case quality with mutation testing approach," 2015 Science and Information Conference (SAI), London, 2015, pp. 902-905.
13. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In CAV'06, 2006.
14. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In ESEC/FSE'05, Sep 2005.
15. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In Proc. PLDI 2005.
16. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In Proc. ESEC/SIGSOFT FSE, 2005.
17. Java Path Finder. https://github.com/javapathfinder/jpf-core/wiki.
18. PITEST (online). www.pitest.org.
19. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing programmer," Computer, vol. 11, no. 4, pp. 34–41, April 1978.
20. R. G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Transactions on Software Engineering, vol. 3, no. 4, pp. 279–290, July1977.
21. Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011.

AUTHORS PROFILE

Avadhani Bindu is a final year M.Tech student in VNR Vignana Jyothi Institute of Engineering & Technology, Hyderabad and was working as an intern in Robert Bosch Engineering And Business Solutions India Ltd, Bengaluru. Her Areas of interest includes Analysis of Algorithms, Machine learning.

Mr. Saumya Ranjan Giri has completed his B.Tech from C.V. Raman College of Engineering, Bhubaneswar and M.Tech from IIIT Bhubaneswar. Currently he is working as a senior engineer in Robert Bosch Engineering & Business Solutions India Ltd with total 8 years of industry experience. His areas of interest includes algorithm analysis and automation. He has been a reviewer of many papers and member of Reviewer Committees in few International conferences. He has many publications in international journals.

Mr. P. Venkateswara Rao, has completed his B.Tech from JNTU Hyderabad, M.Tech from JNTU Hyderabad and has been working as an Assistant professor in VNR Vignana Jyothi Institute of Engineering and Technology, Hyderabad. He is widely known for the teaching profession who acquired 9 years of experience and 2 years of experience research and 1 year in industry. His areas of interest includes machine learning, IOT and Image processing. He has many publications in international journals.