Automated Software Testing Tools

Applications in Large Scale Product Testing

Muhammad Asjad

University of Wollongong

Table of Contents

UNIVERSITY OF WOLLONGONG

Introduction

Automatic test generation exists to save developers from some of the manual work needed to create and maintain test suites[1]. There are both commercial and open source tools available for this purpose. Testing can be classified into system, functional, unit testing etc depending on the level of granularity. There are many techniques and tools created for automated test generation such as testing frameworks, test case generation etc. In this project the focus is on test case generation and how to create effective oracles for real world software. A complete test case consists of both a test input and a test oracle[1].

Unit testing is one of the most common techniques available for creating an oracle. The next issue to be addressed is the creation of effective test cases. "There are many categories of test input generation techniques, such as random testing, symbolic execution, concolic testing, model based testing and search based testing"[1].

This report consists of 3 Main sections. In section 1 we will review some of the tools available for automated testing. In section 2 and 3 we apply some of these tools to real-world software projects. And finally we provide details of the oracle construction and some of the failures detected.

Background

Unit testing is a software verification and validation method in which a programmer tests individual units of code. It is an important component for most regression testing. According to Qu and Robinson[1]unit testing has been around for years but its wide scale industry adoption has been limited. Only blue chip companies like Microsoft have adopted this technology. Most other companies have not developed this practice due to the high cost of creating and maintaining the test suite for the particular software program. The only solution for this is to find some tool to generate these unit tests easily or to aid in its creation and maintenance.

Symbolic execution executes programs with symbolic rather than concrete inputs and maintains a path condition that is updated whenever a branch instruction is executed in order to encode the constraints on the inputs that reach that program point[2].

It is expensive and tedious for testers to manually enter a wide range of inputs to test the software and so there is a need for a tool that can automatically generate inputs to make the job cheaper and less tedious. Concolic testing is one of the techniques that uses a hybrid of random and symbolic testing techniques to generate effective test cases.

Testing Tools Review

In this section we will review about 10 commercially or open source available tools. We will provide a brief description of each tool followed by an evaluation based on our experiences while using the tools.

1.1. Java PathFinder

Web Link: http://babelfish.arc.nasa.gov/trac/jpf/wiki

Type: Symbolic Testing, Concolic Testing

Brief description: JPF is an opensource JAVA testing tool designed for extensibility. It has been around for more than 10 years. It has been used by the research community and NASA for internal testing. JPF started as a software model checker, and was designed to be extensible. Nowadays there are various different execution modes and extensions. JPF generates and analyzes different symbolic execution paths. The input constraints for one path are solved (using off-the-shelf constraint solvers) to generate tests that are guaranteed to execute that path. To bound the search space we put a limit on the model checking search depth, or on the number of constraints along one path[3][7].

Major Features:

- Automated Test case Generation (Symbolic Execution)
- Error checking during test generation process
- Generates an optimized test suite that exercises all the behavior of the system under test.
- Software Model Checking (deadlock and Race Condition detection)
- Deep Inspection (numeric Analysis, invalid access)
- Reports coverage (e.g. MC/DC)

Limitations: The exhaustive searching approach employed by JPF requires a lot of resources (CPU and Memory) and makes exhaustive testing difficult for large projects. JPF provides a lot of configuration options and it is done by writing commands in JPF-configuration file. With the missing GUI, one will find himself consulting the product documentation frequently for even the tiniest configuration. In short, we can say that it is a multi-featured tool and the process of configuration can be quite challenging due to the lack of a GUI.

1.2. Pythoscope:

Weblink: http://pythoscope.org/
Type: Unit Testing

Brief description: Pythoscope is a unit test generator for Python. It has the ability to generate a basic oracle. Pythoscope develops an oracle by building a model of computations that the target program is performing. The tool performs static analysis on the source code and infers rules or conditions that should be met in the program.

Major Features:

- Generates automated unit tests
- Creates a test suite
- Supports automated Oracle Creation

Example Usage: Testing is done primarily by three steps

1. Initialization : This step performs static analysis and creates **.pythoscope to store project.**

    $ pythoscope --init path/to/your/project/

2. Test Case Generation: In this step we will point to the source files of our project and pythoscope will generate test cases.

    $ pythoscope path/to/your/project/specific/file.py path/to/your/project/other/*.py

3. Running the tests:

    This is done by calling unittest.main() inside the generated file and then executed the file containing test cases

Evaluation: Pythoscope is in an experimentation stage of development and stability is not confirmed by the developers. It has a unit test generator and can produce a test suite that can capture the behavior of product under test. Since it is based on unit testing, localizing defects is relatively easier compared to other tools. The oracle generated is usually rudimentary in nature. We have to review all the test cases in order to confirm that the assertions generated by the tool are correct i.e the test case has to be verified against the program specification. Lastly this tool does not have data generator of its own and needs to be integrated to a third party tool in order to generate test cases.

1.3. Nose-parameterized

weblink: https://nose.readthedocs.org/en/latest/index.html

Type: Unit Testing

Brief description: It is built upon nose(which extends Python's unittest). It allows Decorator for parameterized testing with Nose. Nose includes the concept of test functions as generators. If we write our test as a generator that spits out tuples of (callable, argument...), nose will call it and run one test case per yielded value, thus effectively multiplying your number of test cases. We will see will see OKs for values that pass, and one failure per each set of arguments that fails.

Example: Appendix A shows an example of usage.

Features:

- Contains a test runner (which orchestrated the execution of test and summarizes the results.)
- Allows creation of test suite
- Allows Integration of 3rd party data generators
- Automated Test case Generation
- Aggregation of tests into collections
- Independence of the tests from the reporting framework

Usage Summary: This tool its built upon **unittest** framework, and it provides a base class called **TestCase**, which is used to write new test cases. Test suites are implemented by TestSuite Class. This allows individual test and test suites to be aggregated. A test runner is an object that provides a single method, **run()**, which accepts a **TestCase** or **TestSuite** object as a parameter, and returns a result object [1].

Evaluation: It is an excellent tool for constructing and running tests. It is built on top of python's unit-test framework and adds features that overcome its shortcomings. Nose comes with a number of *builtin plugins* to help us output detailed results, perform error introspection, code coverage and doctest etc. Thus making it a comprehensive testing tool for programs written in python. The only drawback of this tool is its reliance on python's unit test framework. According to the documentation "Method Generators are not supported in unittest.TestCase subclasses. Meaning that if our tests are written using unit test.TestCase then we can't rely on this tool.

1.4. DDT (Data-Driven Tests)

Weblink : https://pypi.python.org/pypi/ddt

Brief Description: Allows developers to create multiple test cases from a single method, feeding it a different data using a decorator. This is done by adding a decorator attribute to the test method, which contains the data values it must be run with. The ddt class decorator takes all methods that have been decorated with @data replaces them with as many tweaked copies of themselves as arguments were passed to data. Each tweaked copy is in fact a decorated version of the original method being fed the corresponding test data value.

Example: # tests.py

```
import unittest
from ddt import ddt, data
from mycode import larger_than_two

@ddt
class FooTestCase(unittest.TestCase):
```

```
@data(3, 4, 12, 23)
def test_larger_than_two(self, value):
    self.assertTrue(larger_than_two(value))

@data(1, -3, 2, 0)
def test_not_larger_than_two(self, value):
    self.assertFalse(larger_than_two(value))
```

This will generate following output:

```
----------------------------------------------------------------------
Ran 8 tests in 0.002s

OK
```

Evaluation: From the above example it can been concluded that the number of test cases actually ruan and reported separately has been multiplied. This tool has similar features compared to the previously discussed Nose and is well supported with an update to date documentation available. Using DDT is pretty straight forward and learning curve is not steep. The main drawback of the tool is that it does not include a test runner. And we have to integrate 3rd party tools such as Nose to execute tests in a test suite.

1.5. Randoop:

Weblink: http://mernst.github.io/randoop/

Brief Description

Randoop generates unit tests using feedback-directed random testing, a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs"[4]. Randoop builds inputs and checks them against a set of contracts and filters. The result of the checks determines whether the input is redundant, illegal, contract violating or useful for generating more inputs. Randoop also creates a test suite for the classes under test[5].

Evaluation

Advantages

Compared to other systematic test generation tools,randoops feedback directed random testing managed to detect many previously unknown errors. Also randoop produces fewer redundant tests than most common random testing tools out there. The main limitations found in most open source tools are scalability to large software products and coverage issues that arise when handling pointers and native calls[1]. Randoop does not suffer from scalability issues and can work with large software products. It can automatically generate test cases and an oracle for a given module that can be used for regression testing.

Disadvantages

The tests generated by Randoop are not formatted well in terms of readability. For complex codebases, one has to scrutinize every statement generated in order to properly understand what the generated test function is actually doing. The tool generates a fixed set of inputs for the primitive types, and in order to modify these it needs to be programmatically done or a custom file containing the inputs must be specified. The number of contracts it uses to create unit tests is currently limited. More contracts can be added programmatically. Randoop consumes a lot of resources when run on large software products and can sometimes run out of memory during test case generation.

Table 1

Tool Name	Testing type	Advantages	Disadvantages
Randoop	➢ White-box ➢ Feedback directed Random testing	➢ Automated test case generation ➢ Ease of use ➢ Scalable to large software products ➢ Open Source ➢ Out performs many systematic tools in terms of fault detection ➢ Allows Automated Junit regression Test generation	➢ Poor readability of generated tests ➢ Challenging to control the input values used during test case value generation ➢ Can sometimes runs out of memory and freeze on large codebases. ➢ Sometimes the option to control the location of saved tests failed due to a bug in the eclipse plugin

1.6. CodePro Analytix:

Weblink: https://developers.google.com/java-dev-tools/codepro/doc/

Brief Description

From the documentation CodePro Analytix is the premier Java software testing tool for Eclipse developers who are concerned about improving software quality and reducing developments costs and schedules. The Java software audit features of the tool make it an indispensable assistant to the developer in reducing errors as the code is being developed and keeping coding practices in line with organizational guidelines. The ability to make corrections to the code immediately can dramatically reduce developments costs and improve the speed of finished product delivery.

Code Pro Analytix is a systematic white box testing tool. It generates parameters for methods and determines various combinations.

Evaluation

Advantages

CodePro has an option to automatically generate unit tests in the Junit format for the specified module. It also has the ability to automatically generate inputs for the unit tests making it a full featured automated test case generating tool. It also has many other functions such as code auditing that makes it a complete code analysis tool rather than just a testing tool. The generated unit tests are readable and have documentation in the form of comments for each generated test method. It is easy for beginners to use and is available for free. Another advantage of CodePro is that it supports mock objects which are useful for test driven development.

Disadvantages

CodePro analytix has been abandoned by its developers since 2011 and so does not support the latest versions of the eclipse IDE. All java methods in the JDK have the annotations like the below example

Example 1

```
/**
* Adds the specified element to this set if it is not already
* present.
*
* @param o element to be added to this set.
* @return <tt>true</tt> if the set did not already contain the specified
* element.
*/
```

CodePro is has a feature that supports the design by contract for Java comments as shown in Example 1. But this feature did not seem to work. The dependency graph feature when applied on a large commercial package appears incomprehensible as it has lines scattered all over the graph making it very hard to decipher. Lastly, CodePro generated very few failure detecting tests compared to other tools like Randoop.

Table 2

Tool Name	Testing type	Advantages	Disadvantages
CodePro Analytix	➢ White-box ➢ Systematic Testing	➢ Analysis and reporting for dead code and dependencies ➢ Automated Junit regression test generation ➢ Automated test case generation ➢ Code coverage analysis ➢ Support for	➢ CodePro has not been updated since 2011 ➢ Dependency graph generated is not readable ➢ Design by contract specification option did not work on large codebase

		mock objects ➢ Defect detection and concrete recommendations for fixes	➢ Poor failure detection

1.7. Evosuite

Brief Description

"EvoSuite is a tool that automatically generates test cases with assertions for classes written in Java code. To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behavior"[6].

Evaluation

Advantages:

Evosuite has higher code coverage compared to tools like Randoop. It has an inbuilt technique to narrow down the number of statements/tests generated to those that actually contribute to the code coverage. It artificially introduces faults through mutation testing to in order to check the effectiveness of the generated oracle.

Disadvantages:

It cannot detect out of the ordinary failures like randoop or Jcrasher. Evo suite generates many false positives, and so it can become very hard to maintain generated test cases as they would have to be checked manually with the program documentation to ensure they are correct.

Table 3

Tool Name	Testing type	Advantages	Disadvantages
Evosuite	➢ White-box ➢ Systematic testing(search based)	➢ High code coverage ➢ Generated tests are readable ➢ Uses an evolutionary search technique to narrow down generated tests ➢ Uses mutation testing to create an effective oracle	➢ Generates many false failures

1.8. JCrasher

Weblink : http://ranger.uta.edu/~csallner/jcrasher/

Brief Description

JCrasher is an automatic robustness testing tool for Java code. JCrasher examines the type information of a set of Java classes and constructs code fragments that will create instances of different types to test the behavior of public methods under random data. JCrasher attempts to detect bugs by causing the program under test to "crash"--to throw an undeclared runtime exception. Although in general the random testing approach has many limitations, it also has the advantage of being completely automatic: no supervision is required except for online inspection of the test cases that have caused a crash. Compared to other similar commercial and research tools, JCrasher offers several novelties:

- JCrasher transitively analyzes methods, determines the size of each tested method's parameter-space and selects parameter combinations and therefore test cases at random, taking into account the time allocated for testing
- JCrasher defines heuristics for determining whether a Java exception should be considered a program bug or the JCrasher supplied inputs have violated the code's preconditions
- JCrasher includes support for efficiently undoing all the state changes introduced by previous tests
- JCrasher produces test files for **JUnit**--a popular Java testing tool
- JCrasher can be integrated in the Eclipse IDE

Evaluation

Advantages

It generates data in a truly random fashion and it also generates tests to cover boundary cases. This is useful for creating detecting unexpected scenarios that might occur in a production environment. The process of test case generation is fully automated. It uses heuristics to decide whether the thrown exception is an actual bug of the class or whether it is expected to throw such an exception thus reducing the number of false positives. And finally JCrasher is fully automated, requiring no input from the user other than the subject to be tested.

Disadvantages

JCrasher creates a large number of tests which is hard to look through and review by the developer. Due to its randomness the code coverage is not as much as systematic tools like evosuite. The tests generated by JCrasher cannot be reused as regression tests. They have no further use if a fault is not detected.

Table 4

Tool Name	Testing type	Advantages	Disadvantages
JCrasher	➢ Combines static and dynamic analysis ➢ Random testing	➢ Generates input data in a random fashion ➢ Generates tests to cover boundary cases ➢ Uses heuristics to reduce the number of	➢ Costly maintenance of tests ➢ Limited by the randomness of generated input values ➢ Tests cannot be used as

		false failures ➢ Fully automated	regression tests

1.9. jPet

Weblink: http://costa.ls.fi.upm.es/pet/pet.php

Brief Description

PET is a research project which aims at automatically generating test cases from Java bytecode programs by relying on the technique of partial evaluation. The system receives as input a bytecode program and a set of optional parameters, including a description of a *coverage criterion*; and yields as output a set of test cases which guarantee that the selected coverage criterion is achieved and, optionally, a *test case generator*.

The test case generation process consists of two phases. (1) The decompilation of bytecode into a Constraint Logic Program (CLP). The advantage of decompiling to CLP is that it handles most of the required constraints for free. (2) The generation of test-case generators from CLP decompiled bytecode. To carry out this phase, PET integrates a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. A unique feature of PET is that the test case generators it produces are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the process from scratch. Another important feature of PET is that by using different control strategies of the partial evaluator we can obtain different degrees of coverage

Evaluation

Advantages

jPet generates tests from the byte code rather than the source code. This means you do not need the source code to generate tests. This becomes interesting from the point of view of reverse engineering. When jPet detects a fault it becomes easy to trace the fault back to its origin. This is one of the key advantages of jPet as compared to other tools.

Disadvantages

jPet is platform dependent and works only with linux. The code coverage of jPet is not very high.

Table 5: *refer to next page*

Tool Name	Testing type	Advantages	Disadvantages
jPet	White-box (Unit testing)	➢ Generates tests from the byte code ➢ Easy to trace the location of detected faults ➢ Fully automated ➢ Works with multiple programming languages	➢ Works only with the linux Platform ➢ Medium level code coverage

1.10 Pex

Brief Description

Pex is a systematic testing tool that uses a technique called dynamic symbolic execution. It uses an automatic constraint solver to determine suitable values for the method under test. Pex automatically generates test suites for .Net programs. Right from the Visual Studio code editor, Pex finds interesting input-output values of your methods, which you can save as a small test. Microsoft Pex is a Visual Studio add-in for testing .NET Framework applications.

Evaluation

Advantages

It is fully automated and easy to use. It comes as a plugin for visual studio. The tests generated have a high code coverage. The values generated are intelligent ones obtained through a constraint solver technique.

Disadvantages

Pex can't handle multihreaded programs well. It is restricted to visual studio and .NET Platform. Furthermore cannot handle pointers or arbitrary memory accesses well.

Table 6

Tool Name	Advantages	Disadvantages	Testing type
Pex	➢ Provides Fully Automated test case generation ➢ Generates intelligent test values ➢ High code coverage	➢ Does not work well with multithreaded programs ➢ Does not work well with pointers ➢ Does not work with new versions of visual studio ➢ Its dependent on platform and IDE	➢ Systematic Unit Testing ➢ White box

Tools Summary:

Following table presents the summary of all the above reviewed tables

Table 7

No.	Tool Name	Type	Language
1.	Java Path Finder - JPF	Symbolic Testing	JAVA
2.	Pythoscope:	Unit Testing	Python
3.	Nose-Parameterized	Unit Testing	Python
4.	DDT (Data Drive Testing)	Unit Testing	Python
5.	Randoop	Feedback directed Random testing	JAVA,.NET
6.	CodePro Analytix	Systematic Unit Testing	JAVA
7.	Evosuite	Systematic Testing	JAVA
8.	Jcrasher	Random Testing	JAVA
9.	JPet	Systematic Unit Testing	JAVA
10.	Pex	Systematic Unit Testing	.NET

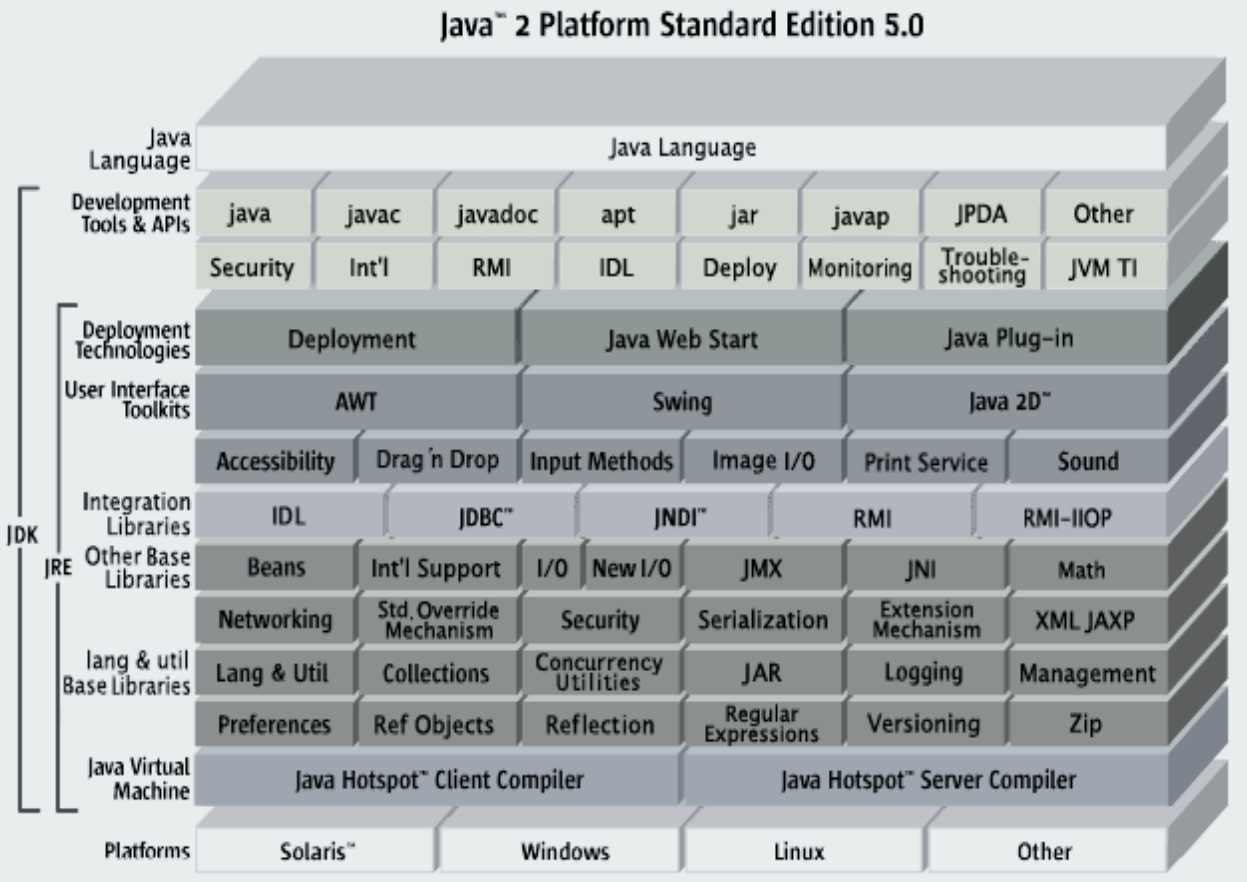
2. Product Description (JDK)

The two principal products in the Java SE platform are: Java Development Kit (JDK) and Java SE Runtime Environment (JRE).

The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications. The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language.

The following conceptual diagram illustrates all the component technologies in Java SE platform and how they fit together. (Source : http://docs.oracle.com/javase/6/docs/)

Diagram 1



JDK is one of the most used SDK's in the world. Over 3 billion devices use Java applications today. The JDK has been released under the GNU General Public License and is currently freely available. The JDK consists of well documented libraries that are regularly updated. We have decided to choose Randoop as the testing tool for this product and apply it on the 1.5 version of the JDK. We chose this tool as it has been well known to detect bugs in various commercial software products including the .NET libraries of Microsoft. We have selected the java.util and java.xml packages from the JDK to be tested.

Table 8

Java Libraries	Lines of Code	Classes	Methods	Description
java.util	39K	204	1019	JAVA Library contains Collections,text,formatting,etc.
Javax.xml	14K	68	437	Libraries for XML processing

3. Oracle Construction for JDK

We use JDK version 1.5 to generate the unit tests and use it for regression testing on version 1.6. Randoop automatically generates an oracle using its inbuilt algorithm shown in the sub section below. We verify the correctness of the oracle by referring to the JDK documentation. As the tests are generated in standard Junit format it is easy to modify the generated oracle to suit the requirements. We run randoop on the java.util and java.xml package shown in table 8.Randoop generates a set of tests for regression testing.

3.1 Randoop Oracle

Randoop allows annotation of the source code to identify observer methods to be used for assert generation.

(Src : Randoop documentation : https://code.google.com/p/randoop/)

The most important component of the Randoop testing tool is the extend operator. The operator *extend(m,seqs,vals)* takes 3 inputs:-

- ***m*** is a method with formal parameters(including the receiver,if any) of type $T_I,\ldots\ldots,T_K$.
- ***seqs*** is a list of sequences of method calls(of the code to be tested)
- ***vals*** nis a list of values $V_I : T_I,\ldots\ldots,V_K : T_K$. Each values is a primitive value or it is the return value of 8.i of the i-th method call for a sequence *s* appearing in *seqs.*

The result of *extend(m,seqs,vals)* is a new sequence that is the concatenation of the input sequences *seqs* in the order that they appear, followed by the method call $m(V_I,\ldots\ldots,V_K)$.

Feedback Directed Random Generation Algorithm:

GenerateSequences(*classes*, *contracts*, *filters*, *timeLimit***)**

1. *errorSeqs* ← {} *// Their execution violates a contract.*
2. *nonErrorSeqs* ← {} *// Their execution violates no contract.*
3. **while** *timeLimit* not reached **do**
4. *// Create new sequence.*
5. m(T**1** . . . T**k**) ← *randomPublicMethod*(*classes*)
6. (*seqs*, *vals*) ← *randomSeqsAndVals*(*nonErrorSeqs*, T**1** . . . T**k**)
7. newSeq ← *extend*(m, *seqs*, *vals*)
8. *// Discard duplicates.*
9. **if** newSeq E *nonErrorSeqs* U *errorSeqs* **then**
10. continue
11. **end if**
12. *// Execute new sequence and check contracts.*
13. (**θ**, *violated*) ← *execute*(newSeq , *contracts*)
14. *// Classify new sequence and outputs.*
15. **if** *violated* = *true* **then**

16. *errorSeqs* ← *errorSeqs* U {newSeq }
17. **else**
18. *nonErrorSeqs* ← *nonErrorSeqs* U {newSeq }
19. setExtensibleFlags(newSeq ,*filters*, **θ**) *// Apply filters.*
20. **end if end while**
21. return (*nonErrorSeqs*, *errorSeqs*)

By default, Randoop uses the following pool of primitive values as inputs to methods:

- byte: -1, 0 1, 10, 100
- short: -1, 0 1, 10, 100
- int: -1, 0 1, 10, 100
- long: -1, 0 1, 10, 100
- float: -1, 0 1, 10, 100
- double: -1, 0 1, 10, 100
- char: '#', ' ', '4', 'a'
- java.lang.String: "", "hi!"

Some of the contracts that *Randoop* checks for are listed below :

- *Equals to null:* o.equals(null) should return false
- *Reflexivity of equality:* o.equals(o) should return true
- *Symmetry of equality:* o1.equals(o2) implies o2.equals(o1)
- *Equals-hashcode:* If o1.equals(o2)==true, then o1.hashCode() == o2.hashCode()
- *No null pointer exceptions:* No NullPointerException is thrown if no null inputs are used in a test.

[4,5]

4. Test Case Generation

Randoop was run on the JDK packages shown in table 8 and a number of test cases were generated. These test cases were then run on Java 1.6 and the following results were obtained.

Table 9

Package Name	Total tests cases generated	Errors detected
Java.util(including sub packages)	6124	18
Java.xml	14,540	0

A total of 18 errors were detected as shown in table 9, some of them being false positives. We used the documentation to separate out the false positives from the failures. We identified 3 genuine failures according to the documentation and have shown them in the code snippets below.

1) Code snippet : **1** shows one of the failures detected by Randoop in the java.util.Date class. The .after() of the date class throws a class cast exception when a date is checked with itself as the parameter. This shows this class has not catered for such a situation where the date object checks itself with the .after() method.

Code Snippet : 1

```
public void test1() throws Throwable {

        /* Constructor of GregorianCalendar
         * GregorianCalendar(int year, int month, int dayOfMonth)
          Constructs a GregorianCalendar with the given date set in the
default timezone with the default locale.
         */

        java.util.GregorianCalendar c1 = new java.util.GregorianCalendar(1000,1,1);

        /*Constructor of Date
         * Date(int year,int month,int date)
          Allocates a Date object and initializes it so that it
represents the time at which it was allocated, measured to the             nearest
millisecond.
         */

        /* getTime method specification
         * public final Date getTime()
            Returns a Date object representing this Calendar's time value (millisecond
            offset from the Epoch").
         */
        java.util.Date d1 = c1.getTime();

        /* public void setSeconds(int seconds)
         * Sets the seconds of this Date to the specified value. This Date
object is modified so that it represents a point in time within             the
specified second of the minute, with the year,month, date,              hour, and
minute the same as before,as interpreted in the local           time zone.
         */
        d1.setSeconds(1);


        /*
         * public boolean after(Date when)
           Tests if this date is after the specified date.
         */

        d1.after(d1);// This statement throws a ClassCastException

   }
```

2) Code Snippet : 2 shows that after the var0 calls the .clone()method, it seems to get modified. This is a failure according to the specification. Both asserts should be true but instead var 14 gets a value of 64 and var 15 gets a value of zero.

Code Snippet : 2

```
public void test1() throws Throwable {

        /*
         * public BitSet()
         * Creates a new bit set. All bits are initially false.
         */
      java.util.BitSet var0 = new java.util.BitSet();

      /*
       * public int size()
       * Returns the number of bits of space actually in use by this
BitSet to represent bit values.
       * The maximum element in the set is the size - 1st element.
       */

      int var14 = var0.size();
      //var 14 gets a value of 64

      assertEquals(64, var14);
      //This assertion passes

      /*This method belongs to the java.lang.Object Class
       * clone()
       * Cloning this BitSet produces a new BitSet that is equal to it.
       */
      java.lang.Object var13 = var0.clone();



      int var15 = var0.size();
      //var 15 gets a value of 0 instead of 64

      assertEquals(64, var15)
      // This assertion fails


  }
```

3) **Code Snippet 3** shows an example of an invalid test generated by Randoop. It passes an illegal parameter that generates an exception. So it is not considered a failure as the class is doing exactly what it is supposed to be doing by throwing an exception.

Code Snippet : 3

```
public void test1() throws Throwable {

    if (debug) System.out.printf("%nRandoopTest_failure_3.test1");


    java.util.NavigableSet var0 = java.util.Collections.emptyNavigableSet();
    /*
     * TreeSet(SortedSet<E> s)
     * Constructs a new tree set containing the same elements and using the same
ordering
     * as the specified sorted set.
     */
    java.util.TreeSet var1 = new java.util.TreeSet((java.util.SortedSet)var0);

    /*
     * add(E e)
     * Adds the specified element to this set if it is not already  present.
     */
    boolean var3 = var1.add((java.lang.Object)(byte)10);

    //1. created emptyNavigableSet()  var0
    //2. created TreeSet Object var1 by passing (SortedSet)var0
    //3. added (Object)(byte)10 to  var1

    int var4 = var1.size();
    // var4 gets the value 1

    java.util.TreeSet var5 = new java.util.TreeSet((java.util.SortedSet)var1);


    //4. created a new treeset obj by passing another treeset obj that holds something
10(typecasted)

    java.util.GregorianCalendar var6 = new java.util.GregorianCalendar();

    int var8 = var6.getActualMaximum(10);
    //var8 gets the value of 11

    java.util.TimeZone var9 = java.util.TimeZone.getDefault();
    java.lang.String var10 = var9.getID();
    var6.setTimeZone(var9);
```

5. F-Measure Calculation

The first failure to be detected when the test suite was run in sequential order was in test case number 2460

F-Measure in sequential order – 2460

Test ID: 5:460

(Note Test ID is in the format File no: Test Method No,

So 5:460 indicates the test method in Project1_JDK > Randoop Test experiments > Randoop Test suite > RandoopTest5.Java : public void test460())

The Test suite consisting of 10,000 test cases was run randomly 1000 times and the F-measure along with the average F-Measure was recorded. This procedure was repeated 4 times giving the following results

Table 10: F-Measure Results

Cycle Number (1000)	Average F-Measure
1	2485
2	2503
3	2465
4	2503

The values of the F-Measure by running the test suite randomly are distributed according to the geometric distribution. If the first error was detected earlier when run sequentially then it would have been considered more effective to do sequential testing for this particular project. But if the first error was detected further down the test suite then random testing would be considered more effective for this project as it detects a failure between the range 2450 and 2510 consistently. So there is no specific evidence to justify one method is better than the other as the results for both in this project are very close. But the sequential order is slightly more effective though by a small margin.

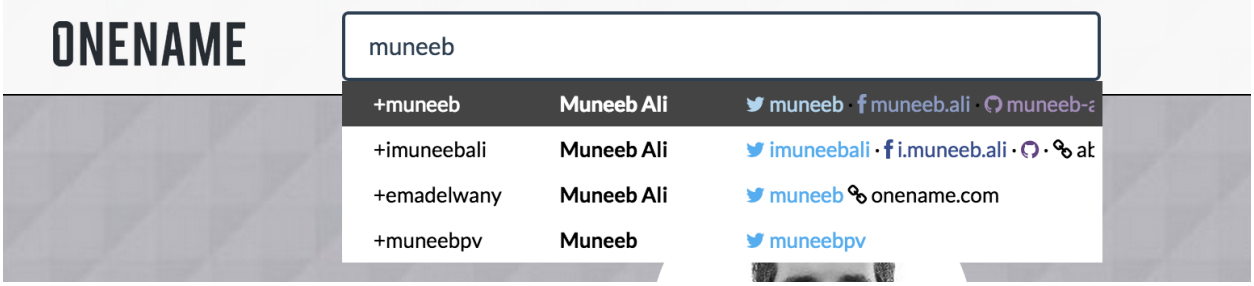
Refer folder Project1_JDK>randoop Test experiments>f-Measure – Test results for complete data

……

```
Iternation number::9972::failure detected at[1928, 4235,
7955]::TestIDs::5:460,17:348,11:252   f-measure: 1928
Iternation number::9973::failure detected at[773, 872,
5166]::TestIDs::11:252,5:460,17:348   f-measure: 773
Iternation number::9974::failure detected at[1682, 5939,
9577]::TestIDs::17:348,11:252,5:460   f-measure: 1682
Iternation number::9975::failure detected at[495, 3331,
5529]::TestIDs::17:348,5:460,11:252   f-measure: 495
Iternation number::9976::failure detected at[2523, 3886,
4506]::TestIDs::11:252,5:460,17:348   f-measure: 2523
Iternation number::9977::failure detected at[74, 4278,
6217]::TestIDs::17:348,5:460,11:252   f-measure: 74
Iternation number::9978::failure detected at[1407, 5593,
6282]::TestIDs::17:348,5:460,11:252   f-measure: 1407
Iternation number::9979::failure detected at[3112, 3653,
8165]::TestIDs::11:252,5:460,17:348   f-measure: 3112
Iternation number::9980::failure detected at[1921, 5051,
9632]::TestIDs::5:460,17:348,11:252   f-measure: 1921
Iternation number::9981::failure detected at[1255, 3128,
4159]::TestIDs::5:460,11:252,17:348   f-measure: 1255
Iternation number::9982::failure detected at[1776, 4251,
4310]::TestIDs::11:252,17:348,5:460   f-measure: 1776
Iternation number::9983::failure detected at[4689, 7758,
9784]::TestIDs::17:348,5:460,11:252   f-measure: 4689
Iternation number::9984::failure detected at[1670, 1691,
6101]::TestIDs::17:348,5:460,11:252   f-measure: 1670
Iternation number::9985::failure detected at[3154, 4682,
8993]::TestIDs::5:460,17:348,11:252   f-measure: 3154
Iternation number::9986::failure detected at[235, 1085,
3405]::TestIDs::5:460,17:348,11:252   f-measure: 235
Iternation number::9987::failure detected at[503, 2385,
5805]::TestIDs::5:460,17:348,11:252   f-measure: 503
Iternation number::9988::failure detected at[870, 2505,
9799]::TestIDs::11:252,5:460,17:348   f-measure: 870
Iternation number::9989::failure detected at[920, 5203,
8745]::TestIDs::5:460,11:252,17:348   f-measure: 920
Iternation number::9990::failure detected at[1185, 1523,
3586]::TestIDs::11:252,5:460,17:348   f-measure: 1185
Iternation number::9991::failure detected at[1157, 6407,
7046]::TestIDs::17:348,5:460,11:252   f-measure: 1157
Iternation number::9992::failure detected at[539, 655,
2043]::TestIDs::5:460,17:348,11:252   f-measure: 539
Iternation number::9993::failure detected at[380, 5570,
8236]::TestIDs::17:348,5:460,11:252   f-measure: 380
Iternation number::9994::failure detected at[2930, 8013,
8937]::TestIDs::11:252,5:460,17:348   f-measure: 2930
Iternation number::9995::failure detected at[1490, 3249,
3263]::TestIDs::5:460,17:348,11:252   f-measure: 1490
Iternation number::9996::failure detected at[2712, 3433,
5166]::TestIDs::17:348,5:460,11:252   f-measure: 2712
Iternation number::9997::failure detected at[9405, 9681,
9871]::TestIDs::11:252,5:460,17:348   f-measure: 9405
Iternation number::9998::failure detected at[622, 2683,
6090]::TestIDs::17:348,5:460,11:252   f-measure: 622
Iternation number::9999::failure detected at[6439, 7774,
8566]::TestIDs::17:348,5:460,11:252   f-measure: 6439

Average f-measure:2485
```

Product – OneName Search



2. Product Description

Web: https://onename.com/

Brief Description: OneName is a decentralized identity and Naming System Built using Bitcoin BlockChain Technology. This System currently has more than 30,000 registered users. A cutting edge search engine has been built for the system that allows users to lookup registered people by Name, Twitter Handle or Bio. One of our team members was involved in the development of this search engine and hence we have had access to its code. This project is under active development and is planned to be made Open Source in the future.

Technologies: The search engine is written is Python. It also employs technologies such as Elastic Search, MongoDB,memcache etc to serve queries to the several thousand of its users.

Tool used: Nose-Parameterized was picked for automated unit testing of the Search Engine's source code. Test Data generation was done by other tools like 'Names' and 'Faker'.

2. Test Case Generation and Oracle Creation Strategy:

In this section we discuss how oracle was created for testing some of the methods in the source code.

Method 1: def valid_username(username)

About: This method is used to validate if the given name is in correct format.. A valid name consists of lowercase letters(for effective matching its internally converted) and digits(at the end).

Oracle: Internally it uses regular expression for pattern matching. The oracle was constructed by making using of two different data generators. One of them generates valid names while the other one generates random data. In each iteration the original method is called twice with valid and invalid values.

Bugs Found: The method returned true and false for valid and invalid names respectively, hence seemed to be working fine. However it was noticed that this method must be modified in case we would like to add a wildcard search feature to the search engine.

Test Case Generation Code:

```
class TestValidUsername(unittest.TestCase):
   @parameterized.expand([(names.get_first_name().lower(),fake.text()) for i in
range(1200)
      ])
   def test_valid_username(self,test_name="default",test_name_gibrish="default deafult"):
      assert_equal(valid_username(test_name),True)

      assert_equal(valid_username(test_name_gibrish),False)
```

Output:

Method 2: def anyword_substring_search_inner(query_word,target_words):

About: from the specifications this method should return True if ANY target_word matches a query_word

Oracle: A random string is generated using data generator. The oracle then selects a substring out of that randomly generated string and passes both(string and substring) to the original method.

Bugs Found: The method executed fine for each of the 1200 attempts. Hence we can say that the original method was correctly implemented.

Test Case Generation Code:

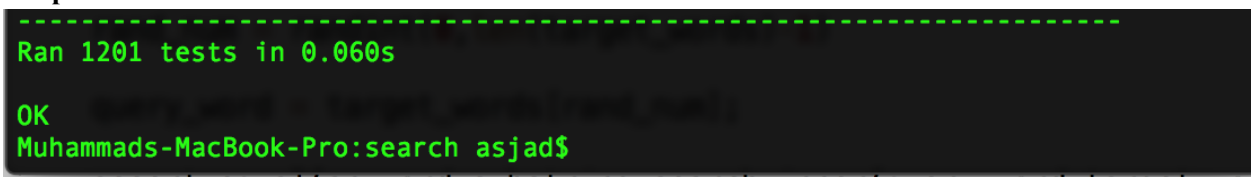
```
class TestAnywordSubstringSearchInner(unittest.TestCase):
   @parameterized.expand([
      (fake.text().lower()) for i in range(1200)])
def test_anyword_substring_search_inner(self,random_text_array="default"):
      target_words = random_text_array.split(' ')

      rand_num = randint(0,len(target_words)-1)

      query_word = target_words[rand_num];

      assert_equal(anyword_substring_search_inner(query_word,target_words),True)
```

Output:



Method 3: anyword_substring_search(target_words,query_words):

About: From the specifications this methods returns True if all query words match.

Oracle: Oracle is constructed similarly to the previous method where a randomly generated string and substring are passed to the original method which validates if the substring is part of the substring or not.

Bugs Found: No bugs were found

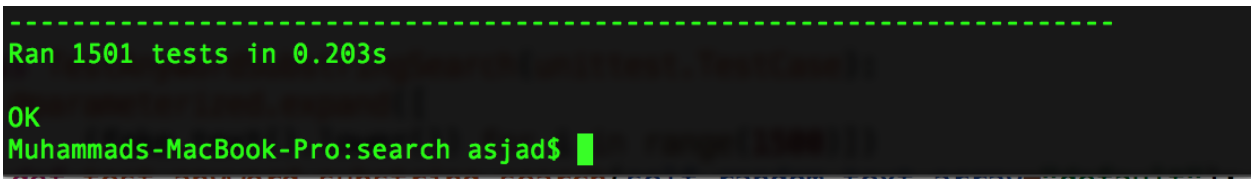
Test Case Generation Code:

```
class TestAnywordSubstringSearch(unittest.TestCase):
    @parameterized.expand([
        (fake.text().lower()) for i in range(1500)])
    def test_anyword_substring_search(self,random_text_array="default"):
        target_words = random_text_array.split(' ')

        query_word = target_words

        assert_equal(anyword_substring_search(target_words,query_word),True)
```

Output:



Method 4: def get_namespace():

About: This function is used during the process creating a search index. An HTTP request is sent to fetch user data stored in blockchain via NameCoin server(accessable internally to the company).

Oracle: Since the URL call to this method was hard-coded inside the method. Initially this method was tested by simply calling it multiple times. Then we made improvements to oracle by adding assertions that check for common sorts of failures that might occurs during a HTTP request.

Bugs Found: A failure was detected in the form of a 'key exception' that the method threw when the call failed. The original method was then corrected by adding a try catch block around the url call to handle the error properly. This allowed us to construct an oracle that checks for all common error that might arise during an HTTP request.

Original Method:

```
def get_namespace():

        url = 'http://ons-server.halfmoonlabs.com/ons/namespace'

        auth_user = 'opennamesystem'
        auth_passwd = 'opennamesystem'

        headers = {'Content-type': 'application/json'}

        r = requests.get(url, headers=headers, auth=(auth_user,auth_passwd))

       results = r.json()['results']

       return results
```

Improved Method to handle failures:

Some research was done to see what sort of code must be added to handling HTTP request failures. It turns out that there are A lot of things can go wrong when requesting information over HTTP from a remote web server: requests timeout, servers fail etc. Following code handles the most common type failures that might occur in a product environment.

Improved Method:

```
def get_namespace():

        url = 'http://ons-server.halfmoonlabs.com/ons/namespace'

        auth_user = 'opennamesystem'
        auth_passwd = 'opennamesystem'

        headers = {'Content-type': 'application/json'}

        try:
                r = requests.get(url, headers=headers, auth=(auth_user,auth_passwd))

                if response.status_code == 503:
                    response.raise_for_status()

                results = r.json()['results']
```

```
except ValueError,KeyError:
            return "KeyError"

except ValueError:
            return "JSON_error"

except requests.exceptions.ConnectionError as e:
            return "connection_error"

except requests.exceptions.ReadTimeout as e:
           return "ReadTimeout"

except requests.exceptions.SSLError as e:
       return "SSLError"

except requests.exceptions.HTTPError as e:
       if e.response.status_code == 503:
            return 503

return results
```

Test Case Generation Code : The final test code then look liked this:

```
class TestGetNamespace(unittest.TestCase):
   def test_get_namespace(self):

       //show Incorrect data return in incorrect format or other error
       assert_not_equal(get_namespace(),"KeyError")
       assert_not_equal(get_namespace(),"JSON_error")

       //Check for Errors connecting to the server
       assert_not_equal(get_namespace(),"connectionerror")
       assert_not_equal(get_namespace()," ReadTimeout")

       //Check for HTTP errors
       assert_not_equal(get_namespace(),503)
```

Output:

```
----------------------------------------------------------------------
Ran 1 test in 31.335s

FAILED (errors=1)
Muhammads-MacBook-Pro:search asjad$
```

Method 5: substring_search(query,list_of_strings,limit_results=DEFAULT_LIMIT)

About: This method takes as input a list of full names and search query(portion of anyname), the Method performs partial matching of the query to see if query is present in the given list of strings.

Oracle: The oracle was constructed by generating a list of names. A random name(first name) is picked out of that list and passed to the substring_search function along with the list. If the name is found

Bugs Found: Initially some false positives were detected because the oracles was not sophistacted enough. Then oracle was improved to detect for partial matching(since it's a feature of search engine) to get rid of the false positives problem.

Final Code:

```
class TestSubstringSearch(unittest.TestCase):
    @parameterized.expand(
        [([[names.get_full_name(gender='female').lower() for i in range(10)]]) for i
in range(50)])
    def test_substring_search(self,names_list=["Muhammad Asjad"]):

        list_length = int(len(names_list))
        list_length = int(list_length - 1)

        rand_num = randint(0,int(list_length))

        search_name = names_list[rand_num]

        temp_name_list = search_name.split(' ')

        search_name = temp_name_list[0]
        result = substring_search(search_name, names_list, 10)
        result =  result[0].split(' ')

        print str(names_list[rand_num])
        print str(result)
        assert_equal(str(search_name), str(result[0]))
```

Output:

```
.
----------------------------------------------------------------------
Ran 1201 tests in 0.106s

OK
Muhammads-MacBook-Pro:search asjad$
```

Testing Summary:

Package Name	Total Tests Cases Generated	Failures detected	False Positives
SubString_Search.py	9,750	4	3

Conclusion

In this project we selected two real-world projects and applied a couple of sophisticated industry testing tools to generate testing tools and to detect failure. In the search Engine product we used unit testing to detect failures and wrote oracles for different methods. Writing the oracle leads to more insights such as identification of external dependencies on other methods or resources. This identification can lead the developer to refactor his code to make it more robust and loosely coupled.

References

1. Xiao, Q. and B. Robinson. *A Case Study of Concolic Testing Tools and their Limitations*. in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2011.
2. Cadar, C., et al., *Symbolic execution for software testing in practice: preliminary assessment*, in *Proceedings of the 33rd International Conference on Software Engineering*. 2011, ACM: Waikiki, Honolulu, HI, USA. p. 1066-1071.
3. Anand, S., C. Păsăreanu, and W. Visser, *JPF–SE: A Symbolic Execution Extension to Java PathFinder*, in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Editors. 2007, Springer Berlin Heidelberg. p. 134-138.
4. Pacheco, C. and M.D. Ernst, *Randoop: feedback-directed random testing for Java*, in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, ACM: Montreal, Quebec, Canada. p. 815-816.
5. Pacheco, C., et al., *Feedback-Directed Random Test Generation*, in *Proceedings of the 29th international conference on Software Engineering*. 2007, IEEE Computer Society. p. 75-84.
6. Fraser, G. and A. Arcuri, *EvoSuite: automatic test suite generation for object-oriented software*, in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, ACM: Szeged, Hungary. p. 416-419.
7. Corina S. Pˇasˇareanu, W.V., *Symbolic Execution and Model Checking for Testing*, in *Haifa Verification Conference*. 2007, Springer. p. 17-18.