# JFuzz: A Tool for Automated Java Unit Testing based on Data Mutation and Metamorphic Testing Methods

Hong Zhu
*Applied Formal Method Research Group*
*Department of Computing and Communication Technologies*
*Oxford Brookes University*
*Oxford OX33 1HX, UK*
*E-mail: hzhu@brookes.ac.uk*

***Abstract***—*Automated test framework plays a significant role in test driven software development methodologies. The XUnit family of testing tools has been widely used in the industry. However, they are weak in supporting test case generation and test result checking. In this paper we propose a new kind of test automation framework by integrating data mutation testing and metamorphic testing methods. A tool for unit testing of Java class called JFuzz is presented. Its uses are illustrated by examples.*

***Keywords****: Software testing, Test Automation Framework, Test Tools, Unit test, Data mutation testing, Metamorphic testing, Fuzz testing, Test driven development*

## I INTRODUCTION

In the past decade, XUnit automated test frameworks have been widely adapted by the industry and plays a significant role in the test driven software development methodology [1, 2, 3]. However, XUnit frameworks provide no support to the generation of test data. It relies completely on the tester to design test cases. Moreover, it also relies on tester to write assert statements to check the correctness of test executions. Consequently, it is observed that, in practice, it is normally that test data are hard coded constants and assertions are only applicable to these constants [4]. Such test cases are so weak that can hardly be considered as a specification of the software. In this paper, we propose a software unit testing tool that aims at improving the automation of unit testing and thus providing a stronger support to test driven software development.

The paper is organized as follows. Section II reviews the data mutation and metamorphic testing methods, and combines them into a new testing method called *mutational metamorphic testing*. It is the methodological foundation of the testing tool JFuzz, which is presented in Section III. Section IV illustrates the uses of JFuzz by examples. Section V concludes the paper with a comparison of the tool with XUnit framework and discusses future works.

## II UNDERLYING TESTING METHODS

JFuzz is developed based on two testing methods data mutation and metamorphic testing. It integrates them into a unified framework. In this section, we briefly overview the testing methods underlying the proposed testing tool.

### *A. Data Mutation Testing*

Data mutation is a test case generation method proposed in [5]. The basic idea is that given a set of test cases, which are called *seeds*, new test cases are generated by modifying the seeds via the applications of a set of operators, which are called *data mutation operators*, or simply *mutation operators*. When the modification of the test data is at random, it also called fuzz testing [6, 7], which has been widely used by the industry, for example, in Microsoft [8, 9], IBM [10], Apple [11], etc.

Similar to program mutation operators, a data mutation operator may be applicable on many different parts of the input data, if the input data are structurally complicated, such as a graph, a trajectory of system parameters, an XML document, a piece of code, etc. In this case, the applicable location of the test data can be considered as an additional parameter of the data mutation operator. Consequently, from a small number of seed test cases, a large number of test cases can be generated by applying a small number of data mutation operators as demonstrated in [5]. Formally, data mutation operators can be defined as follows.

**Definition** 1. (*Data Mutation Operators*)

Let $P$ be the program under test and $D$ be its input domain with an input validity condition $V(x)$. A mapping $F$ from $D^K \times L$ to $D$ is a $K$-ary *data mutation operator* ($K \geq 1$) with parameters in the set $L$, if, for all $x_1, x_2, \ldots, x_K \in D$, $l \in L$, $x_1, x_2, \ldots, x_K$ are valid inputs (i.e. $V(x_i)=True$ for $i=1,2,\ldots, K$) implies that $F(x_1, x_2, \ldots, x_K, l)$ is also a valid input (i.e. $V(F(x_1, x_2, \ldots, x_K, l))$). □

Informally, $V(x)$ means that $x$ is an valid input to program $P$. A $K$-ary data mutation operator takes $K$ valid input data and generates another valid input data according to the value of a parameter $l$.

Figure 1 shows the process of data mutation testing [5].

The following example from [5] illustrates how data mutation testing works. It will also be used later in the paper to explain the proposed new testing method and the uses of tool JFuzz.

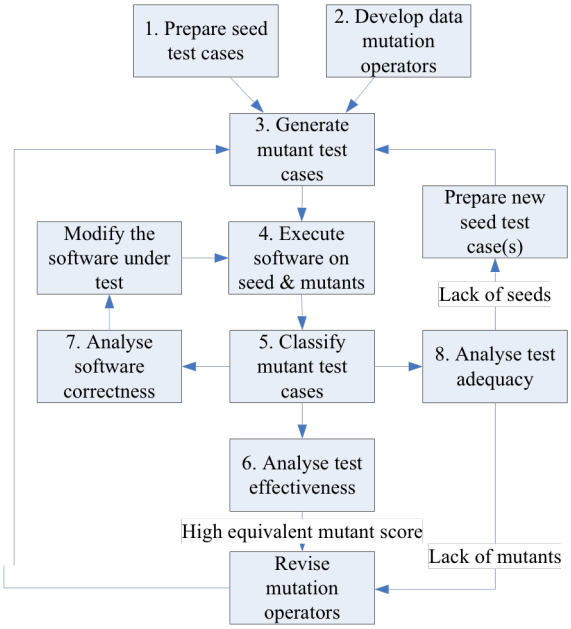


Figure 1. Process of Data Mutation Testing [5]

**Example** 1. Consider a Triangle Classification program whose input consists of three natural numbers $x$, $y$, and $z$ as the lengths of the sides of a triangle. Its function is to classify the triangle into *equilateral* (all sides the same length), or *isosceles* (two the same), or *scalene* (none the same), or to determine that the input does not represent a triangle when the sum of two parameters is not greater than the third.

The following are the seed test cases.

- *Test case* $t_1$: Input: ($x$=5, $y$=5, $z$=5), Expected output: *Equilateral*.
- *Test case* $t_2$: Input: ($x$=5, $y$=5, $z$=7), Expected output: *Isosceles*.
- *Test case* $t_3$: Input: ($x$=5, $y$=7, $z$=9), Expected output: *Scalene*.
- *Test case* $t_4$: Input: ($x$=3, $y$=5, $z$=9), Expected output: *Not a triangle*.

The following are the data mutation operators defined for the Triangle Classification program [5].

- IVP: Increase the value of a parameter by 1;
- DVP: Decrease the value of a parameter by 1;
- SPL: Set the value of a parameter to a very large number, say 1000000;
- SPZ: Set the value of a parameter to 0;
- SPN: Set the value of a parameter to a negative number, say -2;
- WXY: Swap the values of parameters x and y;
- WXZ: Swap the values of parameters x and z;
- WYZ: Swap the values of parameters y and z;
- RPL: Rotate the values of parameters towards left;
- RPR: Rotate the values of parameters towards right.

□

As a part of data mutation testing methodology, a few metrics are defined in [5] to provide guidance for the adequate performance of testing, among which the most important ones include:

- *Seed usage*: the percentage of seeds used to generate mutant test data. A low seed usage indicates that the set of mutation operators is weak and more mutation operators should be defined.
- *Mutation operator usage*: the percentage of mutation operators used in the generation of mutant test data. A low mutation operator usage indicates that the set of seeds is weak and more seeds are needed.
- *Data mutation score*: The percentage of dead mutant test data over the non-equivalent mutants, where is mutant test data is *dead* if it produces an output that is different from the output of the program on the seed. A low mutation score indicates that more seeds and more mutation operators are needed.

Data mutation testing as a test data generation technique a practical and efficient testing method, especially useful for generating test cases for structurally complicated test data. However, an open problem of data mutation testing method is how to enable automatic checking of test results. A solution that we propose here is to integrate data mutation testing with metamorphic testing.

### *B. Metamorphic Testing*

Metamorphic testing was proposed in [12]. It is a test oracle technique and also used to generate test cases. However, it only partially ensures correctness. Here, a test oracle is capable of partially ensuring correctness means that if the program fails a test according to the oracle implies that the program is not correct on the test case. However, if the program passes a test according to the oracle, it does not imply the program is correct on the test case.

The basic idea of metamorphic testing is to use metamorphic relations as the criteria of program correctness. The notion of metamorphic relation can be defined as follows.

**Definition** 2. (*Metamorphic Relations*)

Let program $P$ under test is a function on input domain $D$ and produces output in codomain $C$. Let $K$ be a natural number that $K \geq 2$. A $K$-ary metamorphic relation $M$ is a relation on $D^K \times C^K$ such that program $P$ is correct on input $x_1, x_2, \ldots, x_K$ in $D$ implies that $M(x_1, .., x_K, P(x_1), \ldots, P(x_k))$ holds, where $P(x)$ is program $P$'s output on input $x$. □

The following example from [13] illustrates how metamorphic testing works. We will also use it to explain how our proposed method works.

**Example** 2. A typical example of metamorphic relation for a program that computes $Sin(x)$ function on real numbers is that

$$x_1 + x_2 = \pi \Rightarrow Sin(x_1) = Sin(x_2)\ .$$ □

The metamorphic testing process consists of three steps:

(1) Definition of metamorphic relations that the program should satisfy.

(2) Generation of a test suite $ts_M$ for each $K$-ary metamorphic relation $M$, where each test case $tc$ in the test suite $ts_M$ consists of $K$ input data $x_1, \ldots, x_K$ that satisfy the applicability condition $R$ of $M$.
(3) Execution of program $P$ on each test suite $ts_M$ and check if the program is correct on each test case with regard to the metamorphic relation $M$, i.e. to check if $M(x_1, .., x_K, P(x_1), \ldots, P(x_k))$ is true.

Empirical studies show that metamorphic testing can achieve high fault detection ability [14]. However, there is a lack of systematic method to develop metamorphic relations, and in lack of generally applicable tools to support metamorphic testing.

In the next subsection, we propose a new approach, called *mutational metamorphic testing*, to develop metamorphic relations by integrating it with data mutation testing.

### C. Mutational Metamorphic Testing

We first define the notion of mutational metamorphic relation as follows.

**Definition** 3. (*Mutational Metamorphic Relations*)

Let $P$ be the program under test, $D$ and $C$ be its input domain and output codomain, respectively. Let $f$ be a $K$-ary data mutation operator on $D$ with applicability condition $V(x_1, \ldots, x_K)$ and location parameter $L$. A $K$-ary *mutational metamorphic relation derived from the data mutation operator* $f$ is a relation $R$ on $C^{(K+1)}$ such that the program $P$ is correct on inputs $x_1, \ldots, x_{K+1} \in D$ and $f$ is applicable on $x_1, \ldots, x_K$ imply that $R(P(x_1), \ldots, P(x_{K+1}))$, where $\exists l \in L.\,(x_{K+1} = f(x_1, \ldots, x_K, l))$. □

In other words, a mutational metamorphic relation can be represented in the following form:

$$V(x_1, \ldots, x_K) \Rightarrow R(P(x_1), \ldots, P(x_K), P(f(x_1, \ldots, x_K, l)))$$

**Example** 3. For example, consider the program that computes the *Sin*($x$) function. We define a data mutation operator $f(x)$ on the input domain of real numbers as follows.

$$f(x) = \pi - x.$$

Since this data mutation operator has no applicability constraints and has no location parameter, a mutational metamorphic relation derived from the above mutation operator is that

$$P(x) = P(f(x)).$$

□

In mutational metamorphic testing, a test case for a mutational metamorphic relation $R$ derived from a $K$-ary data mutation operator $f$ consists of $K$ valid input data $x_1, \ldots, x_K$. The testing process consists of the following steps.

(1) Generating a set of test cases that comprise of valid inputs to the program as seeds.
(2) For each seed test case $ts$=($x_1, \ldots, x_K$), finding parameters $l \in L$ that are applicable to the test case, and applying data mutation operator $f$ on the test case $ts$ with each applicable parameter $l$ to generate test data $x_{K+1}$. That is, $x_{K+1} = f(x_1, \ldots, x_K, l)$.
(3) Executing program $P$ on all test data $x_1, \ldots, x_{K+1}$, and record the outputs $P(x_1), \ldots, P(x_K), P(x_{K+1})$.
(4) Checking whether the correctness condition below is satisfied or not:

$$R\big(P(x_1), \ldots, P(x_K), P(x_{K+1})\big).$$

If not, a bug in the program is detected.

Note that, a data mutation operator may use a random value to change the seed. In this case, the data mutation operator is a fuzz operator. Therefore, the testing method proposed here is a generalization of fuzz testing. Moreover, when the applicability condition of the data mutation operator is trivial, i.e. constantly *True* for all input data, the seed test cases can be generated at random, too. The difference is that mutational metamorphic testing uses a metamorphic relation to check test correctness.

**Example** 4.

Consider the data mutation operators defined in Example 1. We now define the mutational metamorphic relation for each of the above data mutation operators.

$$P(t) = Equilateral \Rightarrow P\big(IVP(t)\big) = Isosceles \vee P\big(IVP(t)\big) = nonTriangle$$
$$P(t) = Scalene \Rightarrow P\big(IVP(t)\big) \neq Equilateral$$
$$P(t) = Equilateral \Rightarrow P\big(DVP(t)\big) = Isosceles \vee P\big(DVP(t)\big) = nonTriangle$$
$$P(t) = Scalene \Rightarrow P\big(DVP(t)\big) \neq Equilateral$$
$$P\big(SPL(t)\big) = nonTriangle$$
$$P\big(SPZ(t)\big) = nonTriangle$$
$$P\big(SPN(t)\big) = nonTriangle$$
$$P(t) = P(WXY(t))$$
$$P(t) = P(WXZ(t))$$
$$P(t) = P(WUZ(t))$$
$$P(t) = P(RPL(t))$$
$$P(t) = P(RPR(t))$$

□

## III JFUZZ: A TEST AUTOMATION FRAMEWORK

In this section we present the test automation framework JFuzz, which is a simple tool developed for support mutational metamorphic testing.

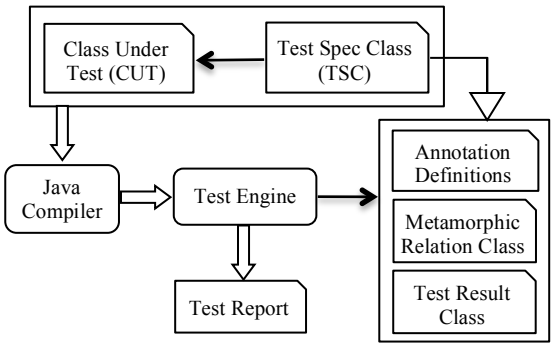


Figure 2. Architecture of JFuzz Test Automation Framework

### A. *The Architecture of JFuzz*

JFuzz is a test automation framework. The inputs to JFuzz are two Java classes: the *class under test* (CUT) and a *test specification class* (TSC), which contains attributes that represent for the seed test cases, methods that are the data mutation operators and methods that are mutational metamorphic relations. The test specification class extends or imports the CUT so that it can access the attributes and methods to be tested. It is compiled before input to the JFuzz tool.

As shown in Figure 2, JFuzz consists of the following components.

- *Annotation Definitions*: These Java classes define a set of annotations that testers can use to annotate the attributes and methods in their Java test code. Three annotations are defined:
  (a) `@Seed` to mark an attribute as a seed test case;
  (b) `@MakeSeed` to mark a method that assigns values to the seeds;
  (c) `@Mutation` to mark a method as creation of mutations to the seeds and invocation of the methods under test and to check the metamorphic relation.
- *Test Result Class*: It defines a set of attributes to record the statistical data of a test, whose values are updated automatically by the Metamorphic Relation Class.
- *Metamorphic Relation Class*: It defines a method called `Assertion`. The `Assertion` method has two parameters: a Boolean value and a String. When the Boolean value is True, the numbers of total mutants and passed mutants are increased by one. When the Boolean value is false, the numbers of total mutants and failed mutants are increased, and the string is output to the test report or print on the screen. An invocation of `Assertion` method implements the mutational metamorphic relation. If the assertion is not satisfied, an error in the CUT is recorded and reported to the tester automatically.
- *Test execution engine*: It performs testing on the CUT according to the TSC and reports the result of testing.

It is worth noting that JFuzz does not directly use the class

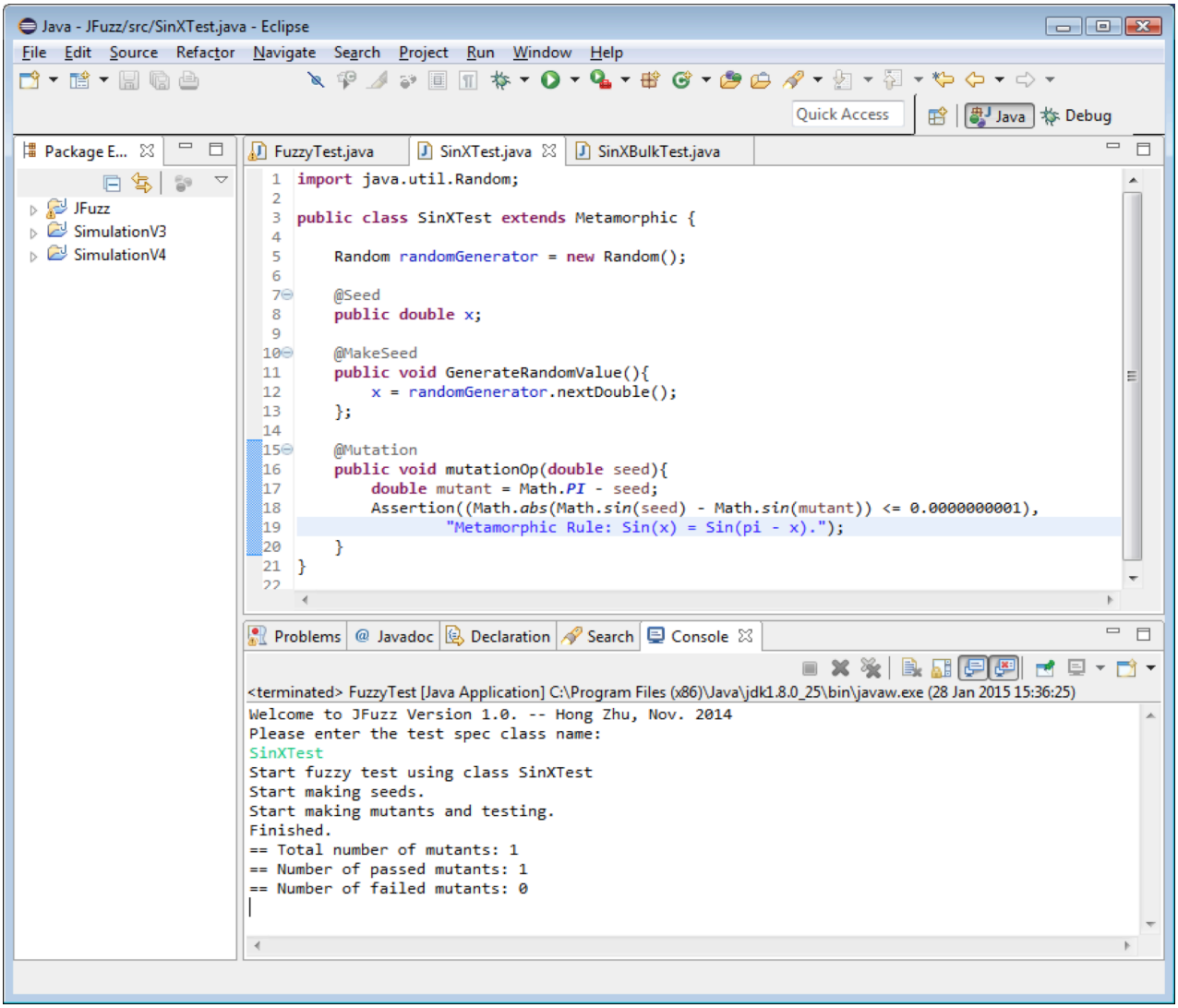


Figure 3. A screen snapshot of executing JFuzz in Eclipse

under test, instead it only executes the methods in a test specification class, and through the test specification class to execute the CUT.

### B. *Test Specification Classes*

A JFuzz test specification class is an ordinary Java class with annotations on the attributes and methods. The annotations used by the test engine are defined in the Annotation definition classes. The following is an example of using these annotations in a test specification class.

**Example** 5. (*Test Specification Class for Testing Sine*)

Figure 4 is an example of the test specification class, which specifies a random testing of the *Sin*($x$) function provided in the Java Math package with a mutational metamorphic relation as the test oracle.

In the example, the attribute "`public double x`" is annotated by "`@Seed`". This means `x` is an attribute that stores a seed test case. In general, there may be multiple seeds as we will see in the examples in Section IV.

The method `public void GenerateRandomValue` is annotated by "`@MakeSeed`". This means `GenerateRandomValue` is a method that creates seeds, or more precisely, it assigns values to the seeds. It is used when the seeds are not constants as in this example. In this example, a random value is assigned to the variable `x`.

The method `public void mutationOp(double seed)` is annotated by "`@mutation`". This means that the method will be applied to the seed test cases one by one to generate mutant test cases. The seed test case is the parameter of the method. A method annotated by `@mutation` should contain code that invokes a method or methods in the CUT on the seed and the mutant test cases, and then to call the `Assertion` method provided by the tool to check the relations between the seed(s) and the mutant(s). In this example, the mutant is a value that equals to $\pi - x$. The assertion states that $|Sin(\pi - x) - Sin(x)| < 10^9$. [*] In general, there may be a number of methods annotated with "`@mutation`" as we will see in the examples given in Section IV. □

Figure 3 is a screen snapshot of the execution of the above test specification class in the Eclipse IDE.

### C. *Test Execution Engine*

The test execution engine of JFuzz is implemented in Java using its reflection and meta-data facilities. Figure 5 gives the algorithm of the test execution engine.

## IV EXAMPLES

In this section, we give two examples of JFuzz test specification classes to demonstrate the style of testing that JFuzz supports.

**Example** 6. (Bulk Testing of the *Sin*($x$) function)

[*] Note that, being floating point numbers, $Sin(\pi - x) = Sin(x)$ may not hold due to round-up error even if the calculation is correct.

```
import java.util.Random;

public class SinXTest extends Metamorphic {

  Random randomGenerator = new Random();

  @Seed
  public double x;

  @MakeSeed
  public void GenerateRandomValue(){
    x = randomGenerator.nextDouble();
  };

  @Mutation
  public void mutationOp(double seed){

    double mutant = Math.PI - seed;

    Assertion(
     (Math.abs(Math.sin(seed)-Math.sin(mutant))
        <= 0.0000000001),
     "Metamorphic Rule: Sin(x)=Sin(pi-x).");
  }
}
```

Figure 4. An Example of JFuzz Test Specification Class

```
Algorithm: Test Execution Engine
  Input:
    Class: ts; // Test specification class
  Output:
    tr: Test Result Report;
  Begin
    //Step 1. Initialization;
    Field[] fs = all Fields declared in ts;
    Method[] ms = all methods declared in ts;
    Create an instance object of the test spec class ts;

    //Step 2. Make seeds;
    for all Methods m in ms do {
        if (m is annotated with "MakeSeed") {
          invoke object tsi's method m;
        }
    };

    //Step 3. Make mutants and perform testing
    for all Fields f in fs {
      if (f is annotated with "Seed") {
        for all Method m in ms {
          if (m is annotated with "Mutation") {
              invoke object tsi's method m
                  with f as parameter;
          }
        }
      }
    };

    //Step 4. Output test result
    Output(tr, tsi's total number of mutants);
    Output(tr, tsi's number of passed mutants);
    Output(tr, tsi's number of failed mutants);

End
```

Figure 5. Algorithm of the Test Engine

In this test specification, we generate 1000 random numbers of `Double` between 0 and 1. These numbers are stored in an array `xs` of `Double` type. Their values are generated by the method `public void GenerateRandomValue()`, which is annotated as `@MakeSeed`.

```
import java.util.Random;
public class SinXBulkTest extends Metamorphic {
   Random randomGenerator = new Random();
   @Seed
   public double[] xs;

   @MakeSeed
   public void GenerateRandomValue(){
     xs = new double[1000];
     for (int i=0; i<1000;i++){
       xs[i]=randomGenerator.nextDouble();
     }
   };
```

The method `public void mutationOp(double[] seed)` below is annotated with the `@Mutation`. It is invoked with the array `xs` as the actual parameter.

```
   @Mutation
   public void mutationOp(double[] seed){
     int num = seed.length;
     double[] mutant = new double[num];
     for (int i=0; i<num; i++){
       mutant[i]= Math.PI - seed[i];
       Assertion((Math.abs(Math.sin(seed[i]) -
         Math.sin(mutant[i])) <= 0.0000000001),
          "Metamorphic Rule: Sin(x)=Sin(pi - x).");
     };
   }
```

Executing this test specification with JFuzz means to perform 1000 random testing on the *Sin*(*x*) function and check the mutational metamorphic relation given in Example 3 with a tolerance of error less than $10^9$ between floating point values. □

The following example is based on the data mutation testing of triangle classification program.

**Example** 7. (Testing Triangle Classification Program)

In Example 1, there are four seed test cases. Thus, we have the following attributes declarations that are annotated as seeds and their values assigned to by the `makeSeed` method.

```
public class TriangleTest1 extends Metamorphic {
   @Seed
   public triangle t1;

   @Seed
   public triangle t2;

   @Seed
   public triangle t3;

   @Seed
   public triangle t4;

   @MakeSeed
   public void makeSeed(){
     t1 = new triangle(5,5,5);
     t2 = new triangle(5,5,7);
     t3 = new triangle(5,7,9);
     t4 = new triangle(3,5,9);
   }
```

There are a number of mutation operators. Each mutation operator is implemented by a Java method. Here we only give the implementations of IPV and the WXY operators. The other mutation operators are very similar.

```
@Mutation
public void IPX(triangle seed){
  System.out.println("---- Mutation IPX on <"
     + seed.x + "," + seed.y + "," + seed.z +">");
   triangle mutant = new triangle(1,1,1);
   mutant.x=seed.x+1;
   mutant.y=seed.y;
   mutant.z=seed.z;
   mutant.Classify();
   if (seed.TriangleType ==
         triangleType.equilaterial) {
    Assertion((
      (mutant.TriangleType ==
         triangleType.isoscelene)
    ||(mutant.TriangleType==
         triangleType.noneTriangle)),
    "Metamorphic Rule for IPX:
      (seed.TriangleType == equilaterial) =>
       ((mutant.TriangleType == isoscelene) or
        (mutant.TriangleType == noneTriangle)");
   };
   if (seed.TriangleType == triangleType.scalene){
    Assertion((
    (mutant.TriangleType !=
       triangleType.equilaterial)),
    "Metamorphic Rule for IPX:
      (seed.TriangleType == scalene) =>
       ((mutant.TriangleType != equilaterial)");
   };
}
```

In the above method, the metamorphic relations for IPD are implemented as two if-statements with two invocations of the `Assertion` method.

The mutational metamorphic relation for WXY is given below as another method annotated with `@Mutation`.

```
@Mutation
public void WXY(triangle seed){
  System.out.println("---- Mutation WXY on <"
     + seed.x + "," + seed.y + "," + seed.z +">" );
  triangle mutant = new triangle(1,1,1);
  mutant.x=seed.y;
  mutant.y=seed.x;
  mutant.z=seed.z;
  mutant.Classify();
  Assertion((seed.TriangleType ==
       mutant.TriangleType),
    "Metamorphic Rule for WXY:
     mutant.TriangleType == seed.TriangleType");
}
```

When this test specification class is executed with JFuzz, each of the mutation method is invoked on each of the four seed test cases, and the results are checked for whether the mutational metamorphic relations were satisfied. A total of 36 mutants were created.

□

## V CONCLUSION

In this paper we proposed the *mutational metamorphic testing method*, which integrates data mutation testing and metamorphic testing methods. The basic idea is to use the data mutation operators as the foundation to derive and express metamorphic relations. It overcomes the shortfalls of these testing methods and retains the advantages of both methods. In particular, it enables test cases to be generated more easily and efficiently and also to enable checking the correctness of test results easily. A nice consequence of the integration is that when a metamorphic relation is universally applicable to all input data, there is no need to have seed test cases. Instead, test cases can be generated at random as we demonstrated in this paper. In that case, mutational metamorphic testing work like fuzz testing, hence the name of the testing tool JFuzz presented in this paper. In contrast, a testing tool that support the general metamorphic testing method has to rely on constraint solver to generate test cases that satisfy the input constraints; see for example, [15].

There are a number of testing tools that supports fuzz testing by generating various types of random data; see, for example, [16]. However, fuzz testing tools does not support test oracles. It only detects faults when the system under test crashes. Mutational metamorphic testing proposed in this paper is much more powerful and effective than fuzz testing tools because it is capable to detect errors more subtle than system crash.

The proposed testing method and tool JFuzz aim to improve unit testing in agile development processes. In comparison with existing test automation frameworks in the xUnit architecture [3, 4], JFuzz provides a stronger support to test case generation and test result checking. Most importantly, the testing method encourages programmers and testers to think not only about known constant test cases and to specify them as the seeds, but also to think about how the input data can be varied and the consequences of the changes in the input data on the program's output and to specify them as mutation operators and the mutational metamorphic relations. Therefore, the test specification is more general and resilient to code changes in the evolution process of agile development. In other words, test specification is closer to the real specification of the software than xUnit style test code.

We are now conducting experiments and case studies to evaluate the effectiveness and efficiency of mutational metamorphic testing method using JFuzz tool. It is worth noting that the notion of mutational metamorphic relations can be further extended, for example, to involve multiple data mutants.

## ACKNOWLEDGEMENT

The author would like to thank Prof. T.Y. Chen at the Swinburne University of Technology, Australia, for his valuable comments on an earlier draft version of the paper.

## REFERENCES

[1] Kent Beck, *Extreme Programming Explained: Embrace Change,* Addison-Wesley, 2000.

[2] Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.

[3] Paul Hamill, *Unit Test Frameworks*, O'Reilly, 2005.

[4] Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

[5] Lijun Shan and Hong Zhu, Generating Structurally Complex Test Cases by Data Mutation: A Case Study of Testing an Automated Modelling Tool. *The Computer Journal*, Vol. 52, No. 5, pp571-588, Aug. 2009.

[6] Michael Sutton, Adam Greene, and Pedram Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[7] Ari Takanen and Jared D. Demott, *Fuzzing for Software Security Testing and Quality Assurance*, Artech House, 2008.

[8] John Neystadt, *Automated Penetration Testing with White-Box Fuzzing*, Microsoft Corporation, Feb., 2008. (Online at: https://msdn.microsoft.com/en-us/library/cc162782.aspx. Last Access: 7 Feb. 2015.)

[9] Patrice Godefroid, Adam Kiezun and Michael Y. Levin, Grammar-based Whitebox Fuzzing, *Proc. of PLDI'08*, June 7–13, 2008, Tucson, Arizona, USA.

[10] Elliotte Rusty Harold, *Fuzz testing: Attack your programs before someone else does*, IBM DeveloperWorks, 26 Sept. 2006. (Online at: http://www.ibm.com/developerworks/library/j-fuzztest/index.html. Last Access: 7 Feb. 2015).

[11] Andy Hertzfeld, *The original Macintosh: Monkey Lives*, Folklore.org, 22 Feb. 1999. (Available Online at http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt. Last access: 7 Feb 2015)

[12] Tsong Yueh Chen, Shing Chi Cheung, and Shiu Ming Yiu, *Metamorphic Testing: A New Approach for Generating Next Test Cases*, Technical Report HKsUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[13] Tsong Yueh Chen, Tsun Him Tse and Zhiquan Zhou, Fault-based Testing Without the Need of Oracles, *Information and Software Technology*, Vol. 45, No. 1, pp1-5, 2003.

[14] Huai Liu, Fei-Ching Kuo, Dave Towey, Tsong Yueh Chen, How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering*, Vol. 40, No. 1, pp4-22, Jan., 2014.

[15] Arnaud Gotlieb and Bernard Botella, Automated Metamorphic Testing, *Proc. of COMPSAC 2003*, pp34-40, IEEE, 3-6 Nov., 2003.

[16] Patrice Godefroid, Michael Y. Levin, and David Molnar, Automated Whitebox Fuzz Testing, *Proc. of NDSS 2008*, 8 - 11 Feb., 2008.