

Morphy: A Datamorphic Software Test Automation Tool

Hong Zhu, Ian Bayley

School of Eng., Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK
Emails: hzhu@brookes.ac.uk,
ibayley@brookes.ac.uk

Dongmei Liu, Xiaoyu Zheng

School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing 210094, China
Email: dmliukz@njust.edu.cn,
zxy961120@sina.com

9 December 2019

Abstract

This paper presents an automated tool called *Morphy* for datamorphic testing. It classifies software test artefacts into test entities and test morphisms, which are mappings on testing entities. In addition to datamorphisms, metamorphisms and seed test case makers, Morphy also employs a set of other test morphisms including test case metrics and filters, test set metrics and filters, test result analysers and test executors to realise test automation. In particular, basic testing activities can be automated by invoking test morphisms. Test strategies can be realised as complex combinations of test morphisms. Test processes can be automated by recording, editing and playing test scripts that invoke test morphisms and strategies. Three types of test strategies have been implemented in Morphy: datamorphism combination strategies, cluster border exploration strategies and strategies for test set optimisation via genetic algorithms. This paper focuses on the datamorphism combination strategies by giving their definitions and implementation algorithms. The paper also illustrates their uses for testing both traditional software and AI applications with three case studies.

Table of Contents

Abstract	i
Table of Contents.....	ii
1. Introduction	1
2. Extended Datamorphic Testing Framework	2
2.1. Test Entities	2
2.2. Test Morphisms	2
2.3. Test Specification	3
3. Test Tool Morphy	3
4. Mutant Combination Strategies	4
5 Case Studies	7
5.1 Triangle Classification	7
5.2. Trigonometric Functions	9
5.3. Face Recognition	10
6. Conclusion	12
6.1. Main Contributions.....	13
6.2. Related Work.....	13
6.3. Future Work.....	14
References	14
Appendix A. The Source code of TestCase and TestPool.....	16
Appendix B. Example of Morphy Test Specification	20
Appendix C. Template of Morphy Test Specification.....	22

Morphy: A Datamorphic Software Test Automation Tool

Hong Zhu, Ian Bayley

School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK
Email: (hzhu|ibayley)@brookes.ac.uk

Dongmei Liu, Xiaoyu Zheng

School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing 210094, China
Emails: dmliukz@njjust.edu.cn, zxy961120@sina.com

Abstract

This paper presents an automated tool called Morphy for datamorphic testing. It classifies software test artefacts into test entities and test morphisms, which are mappings on testing entities. In addition to datamorphisms, metamorphisms and seed test case makers, Morphy also employs a set of other test morphisms including test case metrics and filters, test set metrics and filters, test result analysers and test executors to realise test automation. In particular, basic testing activities can be automated by invoking test morphisms. Test strategies can be realised as complex combinations of test morphisms. Test processes can be automated by recording, editing and playing test scripts that invoke test morphisms and strategies. Three types of test strategies have been implemented in Morphy: datamorphism combination strategies, cluster border exploration strategies and strategies for test set optimisation via genetic algorithms. This paper focuses on the datamorphism combination strategies by giving their definitions and implementation algorithms. The paper also illustrates their uses for testing both traditional software and AI applications with three case studies.

1. Introduction

With the rapid growth of artificial intelligence (AI) in computer applications, ensuring the quality of software components that employ AI techniques becomes indispensable to software engineering. However, testing AI applications is notoriously difficult and prohibitively expensive [10]. It is highly desirable to advance software test automation techniques that meet the requirements of testing AI applications.

Datamorphic testing has been proposed recently as an approach to software test automation [29]. In this method, test automation focuses on the development and application of three types of test code. *Seed makers* generate test cases.

Datamorphisms transform existing test cases into new ones. *Metamorphisms* assert the correctness of test cases. Experiments [29, 30, 19, 3] have demonstrated the it effective at testing AI applications.

However, while datamorphic testing activities can be automated by writing project specific test code, it is highly desirable to develop a general testing tool to achieve the following requirements of test automation.

1. *Reusability* of test code of datamorphisms, metamorphisms, seed makers, etc, even across different projects.
2. *Composability* of test code in different combinations to conduct different experiments with the software under test.
3. *Constructability* of users' own test automation processes from existing test code so that the testing process can be repeated.

To achieve these goals, this paper extends the datamorphic testing framework by introducing the notion of test morphisms and presents an automated software test tool called *Morphy*¹. It enables test automation at three levels. At the lowest level, various test activities can be performed by invoking test morphisms via a click of buttons on Morphy's graphic user interface. At the medium level, Morphy implements various test strategies to perform complicated testing activities through combinations and compositions of test morphisms. At the highest level, test processes are automated by recording, editing and replaying test scripts that consist of a sequence of invocations of test morphisms and strategies.

The paper is organised as follows. Section 2 extends the datamorphic testing framework. Section 3 presents the Morphy test tool. Section 4 defines a set of strategies that combines datamorphisms to generate test data. Section 5 reports three case studies to demonstrate the uses of Morphy. Section 6 concludes the paper by a comparison with related work and a discussion of future work.

¹ Available at <https://github.com/hongzhu6129/MorphyExamples.git>

2. Extended Datamorphic Testing Framework

We extend datamorphic testing method by classifying the software artefacts involved in software testing into two kinds: *entities* and *morphisms*.

Test entities are objects and data used and/or generated in testing, which include *test cases*, *test suites*, the *program under test*, and *test reports*, etc.

Test morphisms are mappings between entities. They generate and transform test entities to achieve testing objectives. They can be implemented by writing test code. They can be invoked to perform test activities and composed to form test processes. Obviously, datamorphisms, metamorphisms and seed makers in the existing model of datamorphic testing are all test morphisms. However, there are other types of test morphisms that play crucial roles in test automation.

A *software test specification* in this extended framework specifies both of these artefacts and enables them to be invoked, composed as well as reused as a test library. In Morphy, a test specification is a Java class that declares a set of attributes for test entities and a set of methods for test morphisms.

In this section, we discuss how they are defined in order to meet the requirements of test automation.

2.1. Test Entities

Test cases and test suites are the most important kinds of entities on which test morphisms are defined. To enable the definition of various test morphisms, a test case must contain not only information about the input and output of the software, but also information about the following:

- How the test case is generated. Two particular pieces of information about the test case are recorded: whether it is a seed or a mutant, and which test morphism generates the test case. In the sequel, the former is called the *feature* of the test case, the latter is called the *type* of the test case.
- How a test case is *related* to other test cases. If a test case is generated by using a datamorphism, the identities of test cases on which the datamorphism applied are recorded, and they are called the *origins* of the test case.
- the *correctness* of the test case. In datamorphic testing, the correctness of a test case is checked against metamorphisms. Each metamorphism can be a partial correctness condition. Therefore, test case may pass some of the metamorphisms but fail on the others. Therefore, the correctness of a test case is a set of records of checking the test case against metamorphisms. We will use the following format to record the correctness:

$\{\text{metamorphismName} : (\text{pass}|\text{fail})\}^*$

A test suite consists of a list of test cases. Each test case is also assigned with a universally unique identifier (UUID). Therefore, the relationships between test cases can be defined by references to their UUIDs.

The Morphy testing tool defines two generic classes *TestCase* and *TestPool* for representing test cases and test suites, respectively. They have two type parameters for the input and output datatypes.

The generic class *TestCase* consists of attributes for (a) the UUID of the test case, (b) the input data, (c) the output data, (d) the feature, (e) the type of the test case, (f) the list of origins, and (g) the correctness of the test case.

The generic class *TestPool* consists of a list of *TestCases* and a number of methods for the operations of the test suite, such as adding and removing test cases to/from the test suite. The test suite used in the testing of the software is declared as an attribute of *TestPool* type and annotated with metadata *@TestSetContainer*. A test specification class can also have attributes and methods without annotations. For examples, an attribute of *TestPool* type without annotation *@TestSetContainer* can be used as an auxiliary test set.

The source code of the *TestCase* and *TestPool* can be found in *Appendix A*.

2.2. Test Morphisms

In addition to the three components of the original datamorphic testing model, we identify the following types of test morphisms that are useful to automate software testing.

- *Test case metrics* are mappings from test cases to real numbers. They measure test cases, for example, on the similarity of a test case to the others in the test set.
- *Test case filters* are mappings from test cases to truth values. They can be used, for example, to decide whether a test case should be included in the test set.
- *Test set metrics* are mappings from test sets to real numbers. They measure the test set, for example, on its quality, such as its code coverage.
- *Test set filters* are mappings from test sets to test sets. A typical example is to remove some test cases from a test set for regression testing.
- *Test executors* execute the program under test on test cases and receive the outputs from the program. They are mappings from a piece of program to a mapping from input data to output. That is, they are functors in category theory.
- *Test result analysers* analyse test results and generate test reports. Thus, they are mappings from test set to test reports.

2.3. Test Specification

A Morphy *test specification* is a Java class, which declares a set of attributes as test entities and a set of methods as test morphisms; see *Appendix B* for an example. Each test morphism is annotated with a metadata to declare the type of test morphism that the method belongs to. Table 1 lists the annotations and datatypes of various types of test morphisms as implemented in Morphy.

Table 1. Annotations of Test Morphisms

Morphism	Annotation	Parameter	Return
Seed Maker	@SeedMaker	Nil	Void
Datamorphism	@Datamorphism	TestCase	TestCase
Metamorphism	@Metamorphism	TestCase	Boolean
Test Case Metrics	@TestCaseMetrics	TestCase	Real
Test Case Filter	@TestCaseFilter	TestCase	Boolean
Test Set Metrics	@TestSetMetrics	Nil	Real
Test Set Filter	@TestSetFilter	Nil	Nil
Test Executer	@TestExecuter	Input	Output
Analyser	@Analyser	Nil	Void

The uses of various types of test entities and morphisms can be found in Section 5.

3. Test Tool Morphy

As shown in Figure 1, Morphy consists of three main facilities: *test set management*, *test runner* and *test scripting*.

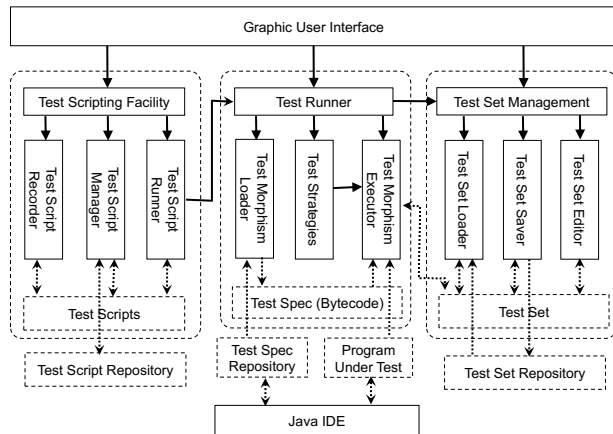


Figure 1. The Architecture of Morphy

The test set management facility enables test sets to be saved into files, loaded from files and edited in a graphic user interface. The test runner enables test specifications to be loaded into the system and various test morphisms of the test specification to be invoked. It also implements various test strategies. The test scripting facility enables interactive testing activities to be recorded as a test script, saved into files, reloaded from files and replayed.

Test specifications can be developed with any Java IDE, but a wizard has been developed as an Eclipse plugin to generate new skeleton Java class as a template of test specification. The skeleton code can be found in *Appendix C*.

Morphy's main graphic user interface, shown in Figure 2, provides a user friendly environment in which testing artefacts can be managed, basic testing activities can be performed and automated testing facilities can be invoked.

At the very top of Morphy's main window are four panels of buttons as follows.

The *Management* panel has buttons for the following functions to manage the artefacts of software testing; these include

- load a Morphy test specification,
- load a previously saved intermediate results of testing from a file,
- save the current state of the testing into a file,
- clean up the system by removing all test cases, messages in the message areas, and the test scripts, etc.

The *Activity* panel has buttons for test activities that invoke various types of test morphisms. These include the following.

- *Seed*: generate seed test cases using selected seed maker methods;
- *Mutate*: generate mutant test cases using selected datamorphisms;
- *Filter*: remove test cases from the current test set using selected test set filters;
- *Edit Test*: show the test cases in the current test pool in a test data window and to enable manual editing of the test results;
- *Measure*: measure the current test set by invoking the selected test set metrics;
- *Execute*: use the TestExecuter method to run testing on test cases in the current test set;
- *Check*: check the correctness of the test results using selected metamorphisms;
- *Analyse*: analyse the test results by invoking the selected analysis method(s).

The *Strategy* panel enables the tester to select from a set of predefined test strategies and to perform automated testing. Three types of test strategies have been implemented in Morphy:

- *Mutant combination*: combining datamorphisms to generate mutant test cases;
- *Domain exploration*: searching for the borders between clusters/subdomains of the input space;
- *Test set optimisation*: optimising test sets by employing genetic algorithms.

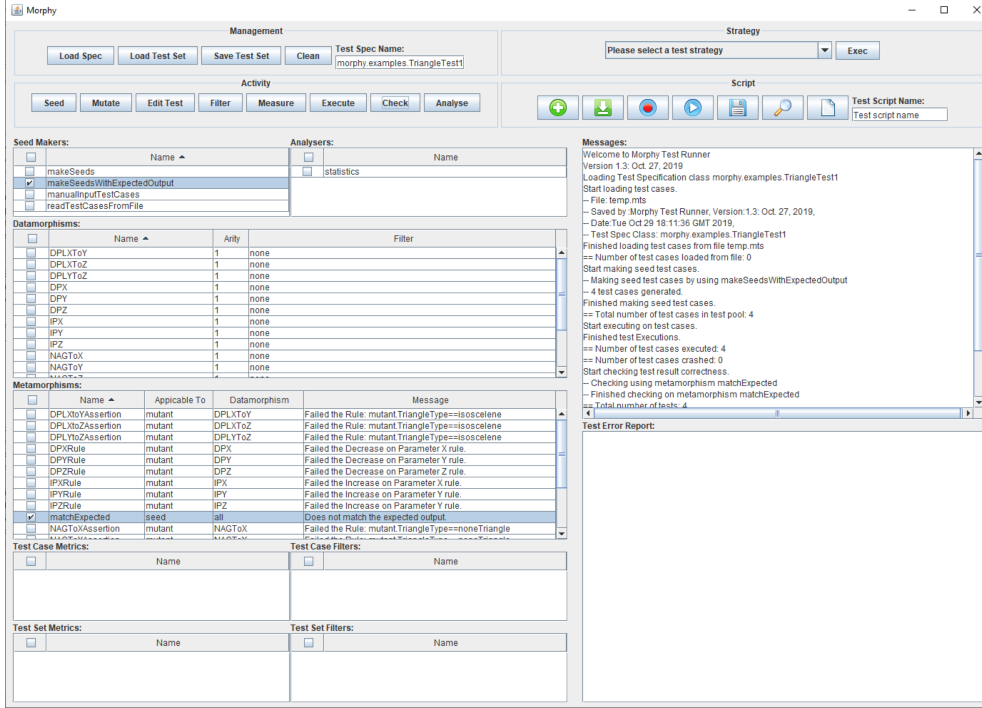


Figure 2. Morphy's Main GUI

Due to the limitation of space, this paper focuses on the mutant combination strategies, which is given in Section 4. The other two types of test strategies will be reported separately.

Finally, the buttons in the *Test Script* panel enable the user to

- record testing activities as a test script,
- view the test script,
- store a test script into a file,
- load a previously stored test script, and
- play a test script.

The test script facility allows the user to automate the testing process. It is particularly useful for repeated testing, such as in regression testing and to obtain data for statistical analysis.

The left-hand side of the main window is a list of tables that shows various types of test morphisms. The elements in these tables can be selected by clicking on the check boxes on the first column of the tables as input to perform the interactive and automatic testing functions mentioned above.

The right-hand side contains two message panels. The upper one reports the testing activities performed and their outcomes. The lower report errors detected by checking the test results against metamorphisms.

Figure 3 shows Morphy's Test Case window. It shows the details of the test cases in the current test set. If test case

metrics are defined in the test specification, the metrics will be applied to the test cases automatically and the results are shown in the table. The test cases in the table can be sorted by a click on the table head row. For example, the test cases in Figure 3 are sorted according to the metrics Distance, shown in column Distance. Test cases can be selected by clicking on the check box on the first column. They can also be selected by applying a test case filter given in the test specification. When the tester clicks on the *Delete* button, the selected test cases will be removed from the table. The *Save* button will then be enabled. The deletion of the test cases from the test set will actually take place when the *Save* button is clicked.

4. Mutant Combination Strategies

Let T be the set of all possible test cases for the software under test. $S \subset T$ be a set of test cases. D be a set of datamorphisms and $d \in D$ is a datamorphism in D . We say that d is k -ary ($k > 0$), if $d : T^k \rightarrow T$.

Definition 1 (First Order Mutants) A test case $y \in T$ is called a first order mutant test case, or simply a first order mutant, of S generated by D , if there is a k -ary datamorphism $d \in D$ and test cases $x_1, \dots, x_k \in S$ such that $y = d(x_1, \dots, x_k)$. \square

[illegible]

Figure 3. Morphy's Test Data Window

Definition 2 (First Order Mutant Completeness) A set C of test cases is first order mutant complete with respect to S and D , if $S \subseteq C$, and for each $d : T^k \rightarrow T \in D$, and each $x_i \in S$, $i = 1, \dots, k$, there is a test case $y \in C$ such that $y = d(x_1, x_2, \dots, x_k)$, where d is k -ary. \square

In other words, a test set is first order mutant complete if it contains every seed and every first order mutant. A test strategy is to test the software with all the seeds and all the first order mutant test cases generated from the seeds using selected datamorphisms.

The following algorithm generates the minimal test set that is first order mutant complete with respect to a give set of seed test cases and a set of datamorphisms.

Algorithm 1 (*Generate 1st Order Mutant Complete Tests*)

```

Input: S = the set of seed test cases;
       D = the set of datamorphisms;
Output: C = a set of test cases;
Variables: tempT = temporal set of test cases;
Begin
1:   C = EmptySet;
2:   for (each datamorphism d in D){
2.1:   tempT = EmptySet;
2.2:   Assume that d is a k-ary datamorphism;
2.3:   forall k-tuples (x1,...,xk) of S {
           add d(x1,...,xk) to tempT;
       };
2.4:   C = C + tempT;
   };
3:   return C + S;
End

```


The following theorem proves the correctness of the algorithm.

Theorem 1 *The test set generated from S using D by Algorithm 1 is the minimal set of test cases that is first order mutant complete with respect to S and D .*

Proof.

(a) *Completeness*: Assume that the output test set C from Algorithm 1 is not complete. This means there is either a seed test case y not in C or there is a first order mutant y generated from seeds $x_1, \dots, x_k \in S$ by using a k -ary datamorphism $d \in D$ is not in C . In the former case, it is in conflict with Step 3. In the latter case, it is in conflict with Step 2.3. Therefore, the assumption is incorrect.

(b) *Minimality*: It is obvious to see that the output only contains seeds and first order mutants. \square

For example, consider a software system that takes a point in the two-dimensional space of real numbers and classifies the points into three subdomains: the *red*, the *blue* and the *black* areas. The test set initially contains 100 random points. The datamorphism is to add the middle point of two test cases. Applying Algorithm 1 produces a first order mutant complete test set, which contains 10000 test cases. Figure 4 (a) and (b) below shows the results of testing on the original 100 random test cases and on the 1st order mutant complete test set, respectively.

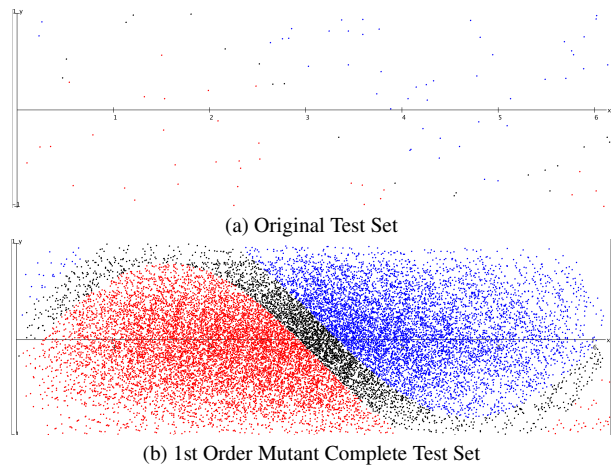


Figure 4. Test Results

Datamorphisms can also be applied to test cases multiple times to generate mutants of mutants, which are called *high order mutants*. For the sake of convenience, a test case $x \in S$ is called a 0'th order mutant of S .

Definition 3 (*Higher order mutants*)

A test case y is a second order mutant of S by D , if there is a k -ary datamorphism $d \in D$ and k test cases x_1, \dots, x_k such that $y = d(x_1, \dots, x_k)$ and for all x_i , x_i is either in S or a first order mutant of S by D , and at least one of x_1, \dots, x_k is a first order mutant of S by D .

A test case y is an n 'th order mutant of S by D ($n > 1$), if there is a k -ary datamorphism $d \in D$ and k test cases x_1, \dots, x_k such that $y = d(x_1, \dots, x_k)$ and x_i are m 'th order mutants of S by D , where $m < n$, and at least one of x_1, \dots, x_k is a $(n-1)$ 'th order mutant of S by D . \square

Similar to first order mutant completeness, a test set is 2nd order mutant complete if it contains all seed test cases, all 1st order mutants and all 2nd order mutant. In general, we have the following definition.

Definition 4 (*K'th order mutant completeness*) A set C of test cases is k 'th order mutant complete with respect to S and D , if it contains all i 'th order mutant test cases of S by D for all $i = 0, \dots, k$. \square

Corollary 1 of Theorem 1. By repeating Algorithm 1 for K times that each time uses the output test set as the input to the next invocation of the algorithm, the result test set is the minimal K 'th order mutant complete.

Proof. By induction on K . Details are omitted. \square

Assume that the set D of datamorphisms contains N methods. If a test set is N 'th order mutant complete with respect to S and D , it contains all permutations of the datamorphisms applied to all test cases. We say that the test set is *permutation complete*. If the datamorphisms are associative, commutative, distributive and idempotent, a permutation complete test set contains all possible test cases that can be derived from a give set of test cases using the set of datamorphisms. The test set is therefore *exhaustive* with regard to the set of seeds and the datamorphisms. It usually contains a huge number of test cases, so the costs of testing could be very high. A compromise is to cover the combinations of datamorphisms.

A mutant of S by D can be represented as a tree on which the leaf nodes are test cases in S , and the non-leaf nodes are datamorphisms in D . The order of a mutant is the height of the tree. Figure 5 below shows some examples of mutants, in which (a) and (b) are first order mutants, and (c) to (f) are second order mutants.

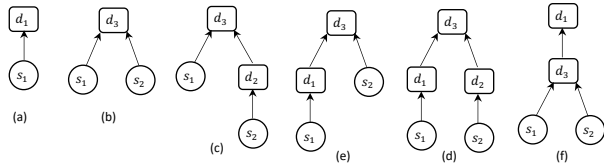


Figure 5. Examples of Mutant Trees

Given a mutant's tree representation, by replacing the test cases associated to the leaf nodes with variables in such a way that each different leaf node is associated with a different variable that ranges over the test cases, we can then obtain a function that generates a high order mutant when substitutes the variables with seed test cases. Each tree of this kind is therefore a way to combine datamorphisms to make higher order mutant test cases from seed test cases. We say that a combination c is k -ary, if it contains k variables, which is equivalent to the number of leaf nodes. We

write $c(x_1, \dots, x_k)$ to represent such a combination of datamorphism. When applying c to seed test cases a_1, \dots, a_k , we write $y = c(a_1, \dots, a_k)$ to denote the result mutant test case. Let $\{d_1, \dots, d_v\}$ be the set of datamorphisms in the tree, we also say that c is a combination of $\{d_1, \dots, d_v\}$. Given a set D of datamorphisms, there may be many different combinations of D .

Definition 5 (*Complete set of datamorphic combinations*) A set C of datamorphism combinations is combinatorial complete for D , if for all subsets $D' \subseteq D$, there is a combination $c \in C$ that contains exactly the datamorphisms in D' . \square

Definition 6 (*Combinatorial complete test sets*) A set V of test cases is combinatorial complete with respect to S and D , if

- there is a set C of datamorphism combinations that is combinatorial complete with respect to D ; and
- for every combination $c \in C$, if c is k -ary, then for all k -tuple of test cases $(x_1, \dots, x_k) \in S^k$, there is a test case y in V such that $y = c(x_1, \dots, x_k)$. \square

The following is an algorithm that generates a combinatorial complete test set.

Algorithm 2 (*Generate Combinatorial Complete Test Set*)

```

Input:  $S$  = the set of seed test cases;
        $D$  = the set of datamorphisms;
Output:  $C$  = a set of test cases;
Variables: tempT = temporal set of test cases;
Begin
1:   for (each datamorphism  $d$  in  $D$ ) {
1.1:   tempT = empty_set;
1.2:   Assume  $d$  is a  $k$ -ary, where  $k > 0$ ;
1.3:   for (all  $k$ -tuples  $(x_1, \dots, x_k)$  of  $S$ ) {
           add  $d(x_1, \dots, x_k)$  to tempT;
       };
1.4:    $S = S + \text{tempT}$ ;
       };
2:   return  $C + S$ ;
End

```

Theorem 2 The test set generated by Algorithm 2 is combinatorial complete with respect to S and D .

Proof. Let C be a subset of D . We write $\|X\|$ to denote the size of a set X . Let $\|D\| = m$. We proof by induction on $n = \|C\| \leq m$ that the test set generated by the algorithm is complete with respect to C .

When $n = 0$: the combination c contains no datamorphism. By step 2, the result test set contains S . Thus, the statement is true in this case.

Assume that when $n = N - 1 < m = \|D\|$ the statement is true.

Now let $n = N \leq m$, and C be any given subset of datamorphisms and $\|C\| = N$. Let d_1, d_2, \dots, d_m be the

sequence of datamorphisms executed in the order through the for-loop in Step 2. Let $\langle d_{i_1}, \dots, d_{i_N} \rangle$ be the order that datamorphisms in C occurred in the for-loop. By the induction assumption, there is a datamorphism combination c' that contains all datamorphisms $d_{i_1}, \dots, d_{i_{(N-1)}}$. Therefore, by Step 2.3 of the algorithm, all permutations of the test cases of the c' as well as all other test cases, including seeds, will be used to generate test cases $d_{i_N}(x_1, \dots, x_k)$. Therefore, the statement is also true for $n = N \leq m$.

By the induction principle, the theorem is true. \square

Note that, the test set generated by Algorithm 2 may be not minimal in size if there is a datamorphism that is non-unity.

5 Case Studies

In this section, we report three case studies on the uses of Morphy in automated software testing.²

5.1 Triangle Classification

Triangle classification is a classic software testing problem that Myer used to illustrate the importance of combination of various types of test cases [20]. The program under test “reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.” [20] Myer listed 14 questions for testers to assess how well he/she tests the program for such a seemingly simple program and reported that “highly qualified professional programmers score, on the average, only 7.8 out of a possible 14”. Here, it is used to demonstrate how to design datamorphisms according to Myer’s assessment criteria to achieve test adequacy and how datamorphic testing method automates the testing process.

5.1.1 The input and output datatypes

Slightly different from the original problem that Mayer specified, the program under test is implemented as a Java class with three attributes x , y and z of integer type and a method *classify* to classify the triangle into four types: *equilateral*, *isosceles*, *scalene*, *noneTriangle*. Therefore, the input data type is a class called *Triangle* with two methods: *toString()* and *valueOf()*, to convert between the object and string representations of a test case. A Java enum class *TriangleType* is declared as the types of triangles. It is the output datatype of test cases.

²The source code of the case studies can be found on GitHub at the URL: <https://github.com/hongzhu6129/MorphyExamples.git>

5.1.2 Generation of seed test cases

There are a number of ways to generate seed test cases in the datamorphic approach. This case study shows how each of them can be easily implemented in a Morphy test specification.

A. Literal Constants. The first is to generate seed test cases as hard coded literal constants and save them into the test pool. The following method in the *TriangleTestSpec* class is a seed maker morphism that generates 4 seed test cases, one in each typical type of triangles.

```
@MakeSeed
public void makeSeeds() {
    testSuite.addInput(new Triangle(5,5,5));
    testSuite.addInput(new Triangle(5,5,7));
    testSuite.addInput(new Triangle(5,7,9));
    testSuite.addInput(new Triangle(3,5,9));
}
```

B. Constants with Expected Output. Expected outputs on seed test cases can also be stored and results of test executions can be checked against the expected outputs. To do so, an additional test pool can be declared to store the expected answers to the seed test cases. The following is a segment of the code in Java.

```
public TestPool<Triangle, TriangleType> expected
    = new TestPool<Triangle, TriangleType>();

@MakeSeed
public void makeSeedsWithExpectedOutput() {
    Triangle trg;
    TestCase<Triangle, TriangleType> tc;

    trg = new Triangle(5,5,5);
    tc = new TestCase<Triangle, TriangleType>();
    tc.input = trg;
    tc.setFeature(TestDataFeature.original);
    testSuite.addTestCase(tc);
    tc.output = TriangleType.equilateral;
    expected.addTestCase(tc);
}
```

A metamorphism can be defined to check the correctness of test executions on seed test cases. The following is such an example.

```
@Metamorphism(
    applicableTestCase="seed",
    message="Match the expected output."
)
public boolean matchExpected(
    TestCase<Triangle, TriangleType> x) {
    String tcId = x.id;
    TestCase expectedTc = expected.get(tcId);
    return (x.output == expectedTc.output);
}
```

C. Manual Input. Seed test cases can be entered into the system through interactive manual input. The following is such an example code.

```

@MakeSeed
public void manualInputTestCases() {
    Triangle trg;
    TestCase<Triangle, TriangleType> tc;
    String numStr;
    while (true) {
        tc = new TestCase<Triangle, TriangleType>();
        trg = new Triangle();
        numStr = JOptionPane.showInputDialog(
            "Please input x:");
        if (numStr==null) { break;}
        trg.x = Integer.valueOf(numStr);
        numStr = JOptionPane.showInputDialog(
            "Please input y:");
        if (numStr==null) { break;}
        trg.y = Integer.valueOf(numStr);
        numStr = JOptionPane.showInputDialog(
            "Please input z:");
        if (numStr==null) { break;}
        trg.z = Integer.valueOf(numStr);
        tc.input = trg;
        tc.setFeature(TestDataFeature.original);
        testSuite.addTestCase(tc);
    }
}

```

D. Read Test Cases from A File. Test cases can also read from a file and then added to the test suite. For the sake of space, here the code is omitted.

5.1.3 Datamorphisms

The main body of the test specification consists of a set of datamorphisms and metamorphisms. The datamorphisms implement Mayers test requirements. For example, Mayer requires that a test set contains all permutations of the three input values so a set of datamorphisms have been designed to generate mutants that are the permutations of the seed test cases. The following is one such datamorphism.

```

@Datamorphism
public TestCase<Triangle, TriangleType> swapXY(
    TestCase<Triangle, TriangleType> seed){
    TestCase<Triangle, TriangleType> mutant
    = new TestCase<Triangle, TriangleType>();
    Triangle m = new Triangle(1,1,1);
    m.x=seed.input.y;
    m.y=seed.input.x;
    m.z=seed.input.z;
    mutant.input = m;
    return mutant;
}

```

Table 1 lists the datamorphisms contained in the test specification. As shown in the above example, these datamorphisms are very simple Java code; each is no more than 10 lines.

Applying these datamorphisms with the first order mutant complete strategy generated 80 mutant test cases, which together with the seed test cases achieved the full coverage of Myer's test requirements.

Table 2. List of Datamorphisms

Name	Function
increaseX	Increase the value of x by 1
increaseY	Increase the value of y by 1
increaseZ	Increase the value of z by 1
decreaseX	Decrease the value of x by 1
decreaseY	Decrease the value of y by 1
decreaseZ	Decrease the value of z by 1
swapXY	Swap the values of x and y
swapXZ	Swap the values of x and z
swapYZ	Swap the values of y and z
rotateL	Rotate the values of x, y and z left
rotateR	Rotate the values of x, y and z right
copyXToY	Copy the value of x to y
copyXToZ	Copy the value of x to z
copyYToZ	Copy the value of y to z
negateX	Negate the value of x
negateY	Negate the value of y
negateZ	Negate the value of z
zeroX	Set the value of x to 0
zeroY	Set of value of y to 0
zeroZ	Set of value of z to 0

5.1.4 Metamorphisms

For each datamorphism, there is a corresponding metamorphism that makes an assertion about the expected output of the program on mutant test cases. For example, for datamorphism *swapXY*, which swaps the values of *x* and *y* of a triangle, the following metamorphism asserts that such a swap will not change the classification outcome.

```

@Metamorphism(
    applicableTestCase="mutant",
    applicableDatamorphism = "swapXY",
    message="Failed the Swap X Y rule."
)
public boolean swapXYRule(
    TestCase<Triangle, TriangleType> x) {
    String originalId = x.getOrigins().get(0);
    TestCase origTc=testSuite.get(originalId);
    return (origTc.output == x.output);
}

```

It is worth noting that annotations restrict the mutant type to which a metamorphism can be applied. For example, the above metamorphism only applies to mutant test cases that are generated by applying the *swapXY* datamorphism.

The metamorphisms in this case study are also very simple; each has no more than 13 lines.

5.1.5 Test Execution and Result Analysis

The test executions of the program under test can easily be defined by a test executor morphism.

```

@TestExecutor
public TriangleType Classifier(Triangle tc) {

```

```

Triangle1 x = new Triangle1(tc.x, tc.y, tc.z);
return x.Classify();
}

```

The analysis of test results can be performed by invoking an analyser morphism. In the case study, an analyser method is written for statistical analysis of test cases and it reports the data in a pop-up. The details are omitted for the sake of space.

Note that test entities and morphisms can be declared in a number of classes to make them more reusable. For example, in this case study, we have put the test executor method in a separate class that inherits the class *TriangleTestSpec*, where the datamorphisms and metamorphisms are declared. Consequently, the test specification class *TriangleTestSpec* can be reused to test a number of different implementations and their versions even if their interface is different. In the case study, we tested two different algorithms for triangle classification, and for each of them, we made two versions, one with errors and one without. The relationships between various classes used in the case study is depicted in Figure 6.

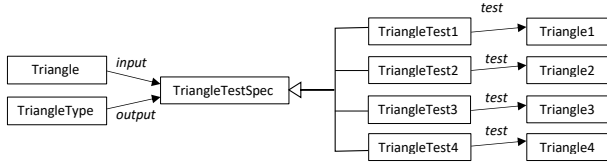


Figure 6. Classes of Test Specification

Using such an organisation for test specifications not only reuses the classes, but also make test scripts reusable without change for testing various different implementations.

5.2 Trigonometric Functions

In this case study, we test three trigonometric functions $\sin(x)$, $\cos(x)$ and $\tan(x)$ provided by Java Math library. The correctness of the library's implementation of these functions will be checked against a set of trigonometric identities and on a set of special values between 0 and 2π as well as random test cases.

5.2.1 Test Cases and Test Suite

The inputs to these trigonometric functions are real numbers and so are the results. However, instead of using *double* or *Double* as the output datatype of test cases, we declare a class called *Trigonometrics* that contains three attributes *sin*, *cos* and *tan* to store the values of these functions and use it as the output datatype. This enables us to check the

functions on identities that involves multiple trigonometric functions.

Two seed maker morphisms are included in the test specification. One generates 20 random inputs in the range from 0 to $\frac{\pi}{2}$, and the other generates a set of special values and the expected output of the functions; see Table 3.

5.2.2 Datamorphisms and Metamorphisms

The datamorphisms for testing trigonometric functions are very simple functions on real numbers; see Table 4. A set of identities of trigonometric functions are implemented as metamorphisms; see Table 5.

The implementations of metamorphisms are straightforward. The following is an example.

```

double error = 0.0000000000001;
@Metamorphism(
    applicableTestCase="mutant",
    applicableDatamorphism="HalfPiMinus",
    message="The rule: sin(pi/2-x) = cos(x)"
)
public boolean HalfPiMinusSinAssertion(
    TestCase<Double, Trigonometrics> tc) {
    TestCase<Double, Trigonometrics> originalTc
        = testSuite.get(tc.getOrigins().get(0));
    return (Math.abs(tc.output.sin
        - originalTc.output.cos) <= error);
}

```

5.2.3 Test Executions and Analysis of Results

The execution of the program on test cases is defined by the following test executor method, which invokes the program under test and stores the output of the program to the test case.

```

@TestExecutor
public Trigonometrics testMath(Double x) {
    Trigonometrics result = new Trigonometrics();
    result.sin = Math.sin(x);
    result.cos = Math.cos(x);
    result.tan = Math.tan(x);
    return result;
}

```

Two analyser methods were used: one for statistical analysis of the test results, reusing the analyser for testing triangle classification program, and the other for visualising the test outputs.

The testing process consists of two stages. The first stage starts with the generation of special value test cases, executions of the functions on them and then using a metamorphism to check whether the output matches the expected value. The analyse of the test results shows that the test detected no error.

The second stage starts by generating 20 random test cases in the range of 0 .. $\pi/2$, then applies the datamorphism $x + \pi/2$ and then $x + \pi$. Then the other datamorphisms are applied to populate the test set. The program

Table 3. The Special Values of Trigonometric Functions Used as Special Test Cases

x	0	$\frac{\pi}{6}$	$\frac{\pi}{4}$	$\frac{\pi}{3}$	$\frac{\pi}{2}$	$\frac{2\pi}{3}$	$\frac{3\pi}{4}$	$\frac{5\pi}{6}$	π	$\frac{7\pi}{6}$	$\frac{5\pi}{4}$	$\frac{4\pi}{3}$	$\frac{3\pi}{2}$	$\frac{5\pi}{3}$	$\frac{7\pi}{4}$	$\frac{11\pi}{6}$	2π
$\sin(x)$	0	$\frac{1}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}}{2}$	1	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{1}{2}$	0	$-\frac{1}{2}$	$-\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{3}}{2}$	-1	$-\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{2}}{2}$	$-\frac{1}{2}$	0
$\cos(x)$	1	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{1}{2}$	0	$-\frac{1}{2}$	$-\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{3}}{2}$	-1	$-\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{2}}{2}$	$-\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}}{2}$	1
$\tan(x)$	0	$\frac{\sqrt{3}}{3}$	1	1	∞	$-\sqrt{3}$	-1	$-\frac{\sqrt{3}}{3}$	0	$\frac{\sqrt{3}}{3}$	1	$\sqrt{3}$	∞	$-\sqrt{3}$	-1	$-\frac{\sqrt{3}}{3}$	0

Table 4. List of Datamorphisms

Name	Function	Name	Function
halfPiPlus	$x \rightarrow \pi/2 + x$	halfPiMinus	$x \rightarrow \pi/2 - x$
piPlus	$x \rightarrow \pi + x$	piMinus	$x \rightarrow \pi - x$
twoPiPlus	$x \rightarrow 2\pi + x$	twoPiMinus	$x \rightarrow 2\pi - x$
sum	$(x, y) \rightarrow x + y$	diff	$(x, y) \rightarrow x - y$
negate	$x \rightarrow -x$		

Table 5. List of Metamorphisms

$\sin(\pi - x) = \sin(x)$	$\sin(\pi + x) = -\sin(x)$
$\cos(\pi - x) = -\cos(x)$	$\cos(\pi + x) = -\cos(x)$
$\tan(\pi - x) = -\tan(x)$	$\tan(\pi + x) = \tan(x)$
$\sin(\pi/2 + x) = \cos(x)$	$\sin(\pi/2 - x) = \cos(x)$
$\cos(\pi/2 + x) = -\sin(x)$	$\cos(\pi/2 - x) = \sin(x)$
$\tan(\pi/2 + x) = -1/\tan(x)$	$\tan(\pi/2 - x) = 1/\tan(x)$
$\sin(2\pi - x) = -\sin(x)$	$\sin(2\pi + x) = \sin(x)$
$\cos(2\pi - x) = \cos(x)$	$\cos(2\pi + x) = \cos(x)$
$\tan(2\pi - x) = -\tan(x)$	$\tan(2\pi + x) = \tan(x)$
$\sin(-x) = -\sin(x)$	$\cos(-x) = \cos(x)$
$\tan(-x) = -\tan(x)$	
$\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$	
$\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$	
$\sin(x - y) = \sin(x)\cos(y) - \cos(x)\sin(y)$	
$\cos(x - y) = \cos(x)\cos(y) + \sin(x)\sin(y)$	
$\tan(x + y) = (\tan(x) + \tan(y))/(1 - \tan(x)\tan(y))$	
$\tan(x - y) = (\tan(x) - \tan(y))/(1 + \tan(x)\tan(y))$	

under test is then executed on the test cases, checked against the metamorphisms, and test results are analysed by invoking the analysers. As shown in Figure 7, the test on random inputs detected a number of errors. The error rate is 0.957%.

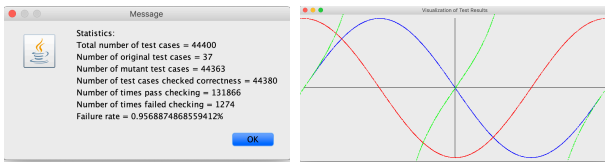


Figure 7. Trigonometric Test Results

Morphy produces an error message when a test case fails a metamorphism. The following is an example of error message in testing the trigonometric functions.

```
-- Rule: tan(pi/2+x) = -1/tan(x) on test case:
```

```
{
  id:09f76c14-8852-404e-9865-fac1e73c63a0,
  input:4.71238898038469,
  output:<-1.0|-1.8369701987210297E-16
  |5.443746451065123E15>,
  feature:mutant,
  type:HalfPiPlus,
  origins:[2b954ce1-ad96-488c-a323-0719065eea72],
  correctness:HalfPiPlusSinAssertion=pass;
    HalfPiPlusCosAssertion=pass;
    HalfPiPlusTanAssertion=fail;
}
```

The error messages produced by Morphy show that the errors all happened when the value of $\tan(x)$ function is used to checking an identity. Two types of errors have occurred: (a) the value of $\tan(x)$ is infinite, e.g. when $x = \pi/2$; (b) the accuracy of an expression is lower than the allowed error, which is 10^{-12} .

5.3. Face Recognition

Face recognition has been employed in the case study on datamorphic testing reported in [29, 30] to demonstrate that datamorphisms can produce meaningful test data for machine learning applications. In this paper, we demonstrate how to achieve test automation and reuses of test code by using Morphy.

5.3.1 Reuses of Test Data

The test data for testing a face recognition application are images of sizes more than 100 KB. In [29, 30], 200 images of different persons were used and each generated 13 mutants using GAN-Attr [13] to alter the facial features. Each image is used to test 4 different face recognition applications. A similar experiment was also conducted to test 5 different face recognition applications [19]. The generation of mutant images from the original images is time consuming. Thus, reusing test data is beneficial.

To enable the reuse of mutant images, the original images and the mutant images are stored in different folders while they have the same file names. Thus, the image input to a face recognition can be represented by a path to the image file in the test case. The datamorphisms can be written simply as manipulation of strings that takes the path to the original image file and produce that of the mutant im-

age file. The output of a face recognition is a real number expressing the similarity between these two images.

The following shows the seed maker and one of the datamorphisms for the experiments reported in [30, 19]. It randomly selects a set of original images. Each test case uses one original image as both the target and subject images. A datamorphism replaces the subject image with a mutant image. Executing a face recognition application on such a mutant test case examines whether the application recognises the person in mutant images as the same person in the target image.

```
@MakeSeed
public void randomSelectOriginalImages() {
    String numStr = JOptionPane.showInputDialog(
        "Entering the number of test cases
        or \"all\":");
    String[] fileList;
    if (numStr==null) { return;}
    if (numStr.equals("all")) {
        fileList = getFileListAll();
    } else {
        int numTestCases = Integer.valueOf(numStr);
        fileList = getFileList(numTestCases);
    }
    for (String name: fileList) {
        TwoImage input = new TwoImage(
            ImageConfig.imagePath,
            image+"\\ "+name, image+"\\ "+name);
        testSuite.addInput(input);
    }
};
```

Two face recognition experiments are reported in [29, 19]. We also designed a third experiment which checks if images of different persons can be distinguished by using both the original images and the mutant images. The following seed maker generates a number of test cases that each test cases consists of two original images selected at random.

```
@MakeSeed
public void randomSelectImagePairs() {
    String numStr = JOptionPane.showInputDialog(
        "Entering the number of test cases:");
    if (numStr==null) { return;};
    int numTestCases = Integer.valueOf(numStr);
    String[] fileList = getFileListAll();
    Random rand = new Random();
    for (int i=0; i<numTestCases; i++) {
        int x = rand.nextInt(fileList.length);
        int y = rand.nextInt(fileList.length);
        String name1 = fileList[x];
        String name2 = fileList[y];
        TwoImage input = new TwoImage(
            ImageConfig.imagePath,
            image+"\\ "+name1, image+"\\ "+ name2);
        testSuite.addInput(input);
    }
};
```

The same set of datamorphisms is applied to these seed test cases using the first order mutant complete strategy. However, the result set of test cases tested a face recognition application on a different property; that is, whether

it can distinguish the different persons when using mutant images if using the original images can. Both experiments can be repeated many times by using the seed makers and the datamorphisms without actually regenerating the mutant images many times. In other words, the datamorphisms and the image data are reused for two different types of experiments.

5.3.2 Test Executions and Analysis of Results

Although the applications under test above have different formats of invocation and return messages, the seed makers, datamorphisms as well as other test morphisms can be reused without any change. The only differences are in the test executors. For each face recognition application, a test executor is written to convert a test case object into the invocation request message and then calls the API. It also receives the return message and converts it into a real number representing the confidence in the face recognition. For example, the following invokes Baidu.com face recognition online service.

```
@TestExecutor
public Double baiduTest(TwoImage images) {
    ApiTesting apiTesting = new BaiduApi();
    String cmd = images.getCmd();
    try {
        String result = apiTesting.similarity(
            cmd.split(" ")[0], cmd.split(" ")[1]);
        return Double.valueOf(result) / 100;
    } catch (Exception e) {
        e.printStackTrace();
        return 0.0;
    }
}
```

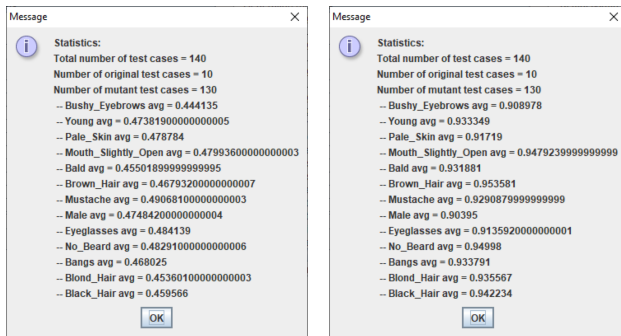
The face recognition software *SeetaFace* is an open source project. The code is written in C++. The project is cloned from the GitHub and installed to the local machine where testing is performed. The invocation of the software is through executing a shell command. The following test executor implements the invocation of *SeetaFace*.

```
@TestExecutor
public Double seetaFaceTest(TwoImage images) {
    String dir = ImageConfig.seetaFacePath;
    String cmd = dir + File.separator
        + "Identification.exe " + images.getCmd();
    try {
        Process proc
            = Runtime.getRuntime().exec(cmd);
        BufferedReader stdout = new BufferedReader(
            new InputStreamReader(
                proc.getInputStream()));
        String result = stdout.readLine();
        stdout.close();
        return Double.valueOf(result);
    } catch (IOException e) {
        e.printStackTrace();
        return 0.0;
    }
}
```

Two analysers were written to analyse the results of the tests. The following calculates the average of the scores on mutant images for each type of mutant and displays the results on screen. The other saves the data to a file.

```
@Analyser
public void viewStatistics() {
    int numTC = testSuite.testSet.size();
    int numOriginalTC = 0;
    int numMutantTC = 0;
    HashMap<String, List<Double>> resMutant
        = new HashMap<>();
    for (TestCase<TwoImage, Double> x :
        testSuite.testSet) {
        if (x.feature == TestDataFeature.original) {
            numOriginalTC++;
        } else {
            numMutantTC++;
            if (!resMutant.keySet().
                contains(x.getType())) {
                resMutant.put(x.getType(),
                    new ArrayList<Double>());
            }
            resMutant.get(x.getType()).add(x.output);
        }
    };
    String message = "Statistics:\n";
    message += "Total number of test cases = "
        + numTC + "\n";
    message += "Number of original test cases = "
        + numOriginalTC + "\n";
    message += "Number of mutant test cases = "
        + numMutantTC + "\n";
    for (String type: resMutant.keySet()) {
        Double avg = resMutant.get(type).stream()
            .mapToDouble(Double::doubleValue)
            .average().getAsDouble();
        message += " -- "+ type+" avg = "+avg+"\n";
    }
    JOptionPane.showMessageDialog(null, message);
}
```

Figure 8 gives the screen snapshots of the above analyser on two experiments with one face recognition application.



(a) Images of different persons

(b) Images of the same person

Figure 8. Face Recognition Test Results

5.3.3 Test Scripts

As a typical AI application, testing face recognition requires repeated experiments in order to obtain results that are statistically significant. Test scripts were written to further improve test automation for repeated experiments. The following is one of the test scripts written in the case study.

```
//Experiment with images of same person
//Testing Baidu.com face recognition
saveMessage(ExpDataFile.csv;
    Baidu, same person)
loadTestSpec(file:C:\Users\Hong\git\Morphy3\bin
    \morphy\examples\FaceRecBaiduTest.class)
loadTestSet(C:\Users\Hong\git\Morphy3\ExpData
    \FaceRecognitionExp1)
executeTestCases()
analyse([statisticalAnalysis])

//Testing FacePlus.com face recognition
saveMessage(ExpDataFile.csv;
    FacePlus, same person)
loadTestSpec(file:C:\Users\Hong\git\Morphy3\bin
    \morphy\examples\FaceRecFacePlusTest.class)
loadTestSet(C:\Users\Hong\git\Morphy3\ExpData
    \FaceRecognitionExp1)
executeTestCases()
analyse([statisticalAnalysis])

//Testing SeetaFace face recognition
saveMessage(ExpDataFile.csv;
    SeetaFace, same person)
loadTestSpec(file:C:\Users\Hong\git\Morphy3\bin
    \morphy\examples\FaceRecSeetaFaceTest.class)
loadTestSet(C:\Users\Hong\git\Morphy3\ExpData
    \FaceRecognitionExp1)
executeTestCases()
analyse([statisticalAnalysis])

//Testing Tencent.com face recognition
saveMessage(ExpDataFile.csv;
    Tencent, same person)
loadTestSpec(file:C:\Users\Hong\git\Morphy3\bin
    \morphy\examples\FaceRecTencentTest.class)
loadTestSet(C:\Users\Hong\git\Morphy3\ExpData
    \FaceRecognitionExp1)
executeTestCases()
analyse([statisticalAnalysis])
//Test script end
```

5.4. Discussion

The following observation were made on the case studies.

First, test morphisms in the case studies are simple and easy to write; see Table 6, where TC stands for Triangle Classification, Trg for Trigonometric Function, and FR for Face Recognition. LOC is the lines of code.

Second, test specifications can be reusable if it is decomposed into a number of classes where common test morphisms are placed together while test specific morphisms are placed in separate classes.

Table 6. Summary of Case Studies

	TC	Trg	FR
Num of Classes	11	4	8
Total LOC	899	830	450
Num of Seed Makers	4	3	3
Average LOC of Seed Makers	26.25	61.67	21.33
Num of Datamorphisms	20	10	13
Average LOC of Datamorphisms	9	6	8
Num of Metamorphisms	25	30	–
Average LOC of Metamorphisms	8.72	7.00	–
Num of Analysers	2	2	2
Average LOC of Analysers	62	33	41

Third, achieving test automation using facilities at three different levels of activity, strategy and process is flexible and practical. Different testing tools and techniques can be easily integrated into the Morphy framework and used together.

6. Conclusion

6.1. Main Contributions

This paper redefines the datamorphic testing method by classifying test artefacts into test entities and test morphisms. The notion of test morphisms is defined as mappings between software test entities. Datamorphisms and metamorphisms are examples of such test morphisms. The former are mappings from test cases to test cases, while the later are mapping from test cases to Boolean values as assertions on the correctness of the test. Seed makers are also test morphisms, which map into test sets. We identified a set of other test morphisms, which include test case metrics and filters, test set metrics and filters, test executors and analysers.

The paper also presents a test automation tool called *Morphy*. We demonstrated that testing activities can be automated by writing test codes for various test morphisms and invoking such test codes through a test tool like Morphy. Advanced combinations of test morphisms can be realised by implementing of test strategies to achieve a higher level of test automation. Three types of test strategies have been implemented in Morphy: (a) Datamorphism combination strategies, which generate test sets of various coverage of datamorphisms combinations; (b) Exploration strategies, which explore the test space in order to find the borders between subdomains for testing classification and clustering type of AI applications; (c) Test set optimisation strategies, which employ genetic algorithms optimise test sets. This paper focuses on the first of these, defining both strategies and implementation algorithms, with the other types of test strategies being reported separately.

Morphy also provides a record-replay type of test automation facility to further improve test automation especially for regression testing. Test scripts can be recorded from the interactive invocations of test morphisms for basic test activities and invocations of test strategies as well as test management activities such as load test specifications and load and save test sets, etc. Although the case studies reported in this paper used test scripts to improve test automation, a more detailed study of the test script facility will be reported in a separate paper.

6.2. Related Work

Morphy is a test automation framework applicable to all kinds of software systems including AI applications as demonstrated in the case studies. In comparison with existing test automation frameworks like XUnit [11, 18] and GUI test script based test tools like Selenium [23] and WebDriver [26], Morphy provides more advanced test automation facilities such as test strategies and test scripting.

In XUnit framework, testing is defined by a set of methods in a class or a set of test scripts for executing the program under test together with methods for setting up the environment before test executions and tearing down the environment after test. Such a test specification is imperative. Our test specification is declaratively imperative: various types of test morphisms are declared in a test specification while each test morphism is coded in an imperative programming language. Our case studies show that such test specifications are highly reusable and composable even for testing different applications. This is what existing test automation frameworks have not achieved.

In a GUI based test automation framework, test automation is realised by test scripts or test code that interact with GUI elements. The most representative and most well-known example of such testing tools is Selenium [23]. Selenium has two test environments: (a) the Selenium IDE in which manual GUI based testing can be recorded into a test script and replayed; (b) the Selenium Web Drivers, which provides an API with methods for web-based GUI elements. Test code can be written in various programming languages that are equivalent to test scripts. Morphy also employs test scripts to achieve a high level of test automation, but it is equipped with more advanced test automation facilities such as test strategies.

An advantage of Morphy is that the architecture enables various testing techniques and tools to be integrated by wrapping exiting testing tools as methods in test specification class that invoke the tools. For example, test case generation techniques and tools [1] like fuzz testing [25], data mutation testing [22], random testing [2], adaptive random testing [8, 17], combinatorial testing [21] and model based test case generators are all test morphisms, which

can be wrapped as seed makers or datamorphisms. Metamorphic relations in metamorphic testing [7] and formal specification-based test oracles [4] are metamorphisms. Algebraic specification has been used for both generating test cases and checking test correctness [5, 6, 28]. The techniques that implements automated testing based on algebraic specifications [4, 15, 16] can easily be split into two parts, the test morphisms to generate test cases and the test morphisms to check test correctness. Test coverage measurement tools like [24] are test set metrics. Regression testing techniques and methods [27] that select or prioritise test cases in an existing test set can be implemented as test set filters. Search-based testing [12, 9] can be regarded as test strategies. Therefore, they can all be easily integrated into Morphy.

6.3. Future Work

It is worth noting that datamorphic testing focuses on test morphisms related to test data and test sets, as its name implies. There are other types of test morphisms. For example, mutation operators in mutation testing [14] and fault injection tools for other fault-based testing methods are test morphisms that are mappings from programs to programs or sets of programs. Specification mutation operators are test morphisms that mapping from formal specifications to specifications or sets of specifications. It is an interesting further research question how to integrate such test morphisms into the datamorphic testing tools like Morphy, although, theoretically speaking, there should be no significant difficulty to do so.

It is also possible to integrate XUnit test automation frameworks like JUnit and GUI based test automation tools like WebDriver with Morphy. This is also an interesting topic for future work.

We have already conducted some experiments with the exploratory strategies and genetic algorithms for test set optimisation strategies. The results will be reported in separate papers.

References

- [1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
- [2] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012.
- [3] Y. Bagge. Experiments with testing a bounded model checker for C. MSc Dissertation, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK, Sept 2019.
- [4] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, pages 387–405, Nov. 1991.
- [5] H. Chen, T. Tse, and T. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM TOSEM*, 7(3):250–295, 1998.
- [6] H. Chen, T. Tse, and T. Chen. Tackle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.
- [7] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, Jan. 2018.
- [8] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proc. of the 9th Asian Computing Science Conf.*, LNCS 3321, pages 320–329. Springer, 2004.
- [9] M. Dave and R. Agrawal. Search based techniques and mutation analysis in automatic test case generation: A survey. In *2015 IEEE International Advance Computing Conference (IACC)*, pages 795–799, June 2015.
- [10] A. Gotlieb, M. Roper, and P. Zhang, editors. *Proceedings of The First IEEE International Conference On Artificial Intelligence Testing (AITest 2019)*. IEEE Computer Society, Los Alamitos, CA, USA, Apr 2019.
- [11] P. Hamill. *Unit Test Frameworks*. O’Reilly, 2005.
- [12] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):Article 11, 61 pages, Nov. 2012.
- [13] Z. He, W. Zuo, M. Kan, S. Shan, and X. Chen. Attgan: Facial attribute editing by only changing what you want. *IEEE Transactions on Image Processing*, 28(11):5464–5478, Nov 2019.
- [14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sep. 2011.
- [15] L. Kong, H. Zhu, and B. Zhou. Automated testing EJB components based on algebraic specifications. In *COMPSAC (2)*, pages 717–722, 2007.
- [16] D. Liu, X. Wu, X. Zhang, H. Zhu, and I. Bayley. Monic testing of web services based on algebraic specifications. In *Proc. of the 10th IEEE International Conference on Service Oriented System Engineering (SOSE 2016)*, Oxford, England, UK, March 2016. IEEE Computer Society.
- [17] Y. Liu and H. Zhu. An experimental evaluation of the reliability of adaptive random testing methods. In *Proc. of The Second IEEE International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008)*, pages 24–31, Yokohama, Japan, July 2008.
- [18] G. Meszarros. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.
- [19] S. Mugutdinov. Applying datamorphic technique to test face recognition applications. BSc dissertation, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford, UK, March 2019.
- [20] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, Inc., 2nd edition edition, 2004.

- [21] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, Feb. 2011.
- [22] L. Shan and H. Zhu. Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. *The Computer Journal*, 52(5):571–588, Aug 2009.
- [23] Software Freedom Conservancy. Selenium website. Online at URL: <https://selenium.dev>, Nov. 2019.
- [24] Stackify. The ultimate list of code coverage tools. Online at URL: <https://stackify.com/code-coverage-tools/>, May 2017.
- [25] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.
- [26] WWW Consortium. WebDriver: W3C Recommendation. Online at URL: <https://www.w3.org/TR/webdriver1/>, June 2018.
- [27] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.
- [28] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *Proc. of QSIC'03*, pages 91–99, Dallas, USA, Nov. 2003.
- [29] H. Zhu, D. Liu, I. Bayley, R. Harrison, and F. Cuzzolin. Datamorphic testing: A method for testing intelligent applications. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest 2019)*, pages 149–156, Los Alamitos, CA, USA, Apr 2019. IEEE Computer Society.
- [30] H. Zhu, D. Liu, I. Ian Bayley, R. Harrison, and F. Cuzzolin. Datamorphic testing: A methodology for testing ai applications. Technical Report OBU-ECM-AFM-2018-02, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK, Dec 2018.

Appendix A. Source Code of TestCase.java and TestPool.java

```
package morphy.annotations;
import java.util.List;
import java.util.UUID;

public class TestCase<InputType,OutputType> {
    /**
     * the id number of the test case;
     */
    public String id;

    /**
     * the input data of the test case
     */
    public InputType input;

    /**
     * the output data of the test case
     */
    public OutputType output;

    /**
     * the feature of the test case, which can be either "original" or "mutant";
     */
    public TestDataFeature feature;

    /**
     * the type of the mutant, which should be the name of the operator use to generate the mutant;
     */
    private String type;

    /**
     * the id numbers of the original test cases that are used to generate the mutant;
     */
    public List<String> origins;

    /**
     * the correctness of the test case. Format: "metamorphismName:(pass|fail);"
     */
    public String correctness;

    /** The constructor of TestCase and it initialises a new test case
     *
     */
    public TestCase() {
        id=UUID.randomUUID().toString();
        input = null;
        output = null;
        setFeature(TestDataFeature.original);
        setType("");
        correctness = "";
        setOrigins(null);
    }

    /** Convert a test case into a string */
    public String toString() {
        String outputStr="{\n";
        outputStr = outputStr + " id:" + id + ",\n";
        if (!(input == null)) {
            outputStr = outputStr + " input:" + input + ",\n";
        }else{
            outputStr = outputStr + " input:\n";
        };
        if (!(output == null)) {
            outputStr = outputStr + " output:" + output + ",\n";
        }else{
            outputStr = outputStr + " output:\n";
        };
        outputStr = outputStr + " feature:" + getFeature().toString() + ",\n";
        outputStr = outputStr + " type:" + getType() + ",\n";
        outputStr = outputStr + " origins:[";
```

```

        if (!(getOrigins() == null)) {
            for (int i=0; i<getOrigins().size(); i++) {
                outputStr = outputStr + " " + getOrigins().get(i).toString();
                if (i!=getOrigins().size()-1) {
                    outputStr = outputStr + ",";
                }
            }
            outputStr = outputStr + "],\n";
            outputStr = outputStr + " correctness:" + correctness + "\n";
            outputStr = outputStr + "]\n" ;
            return outputStr;
        }

    public static TestDataFeature strToFeature(String featureStr) {
        switch (featureStr) {
            case "original": return TestDataFeature.original;
            case "mutant" : return TestDataFeature.mutant;
            default: return null;
        }
    }

    public List<String> getOrigins() {
        return origins;
    }

    public void setOrigins(List<String> origins) {
        this.origins = origins;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public TestDataFeature getFeature() {
        return feature;
    }

    public void setFeature(TestDataFeature feature) {
        this.feature = feature;
    }
}

package morphy.annotations;
import java.util.List;
import java.util.ArrayList;

public class TestPool<InputType,OutputType> {
    /**
     * container to hold a test set;
     */
    public List<TestCase<InputType,OutputType>> testSet;

    /**
     * Constructor to create a new empty set pool
     */
    public TestPool() {
        testSet = new ArrayList<TestCase<InputType,OutputType>>();
    }

    /**
     * Add a new test case with a value as the input. The new test case is an original (seed) test case.
     * @param v the value of the input
     */
    public String addInput(InputType v) {
        TestCase<InputType,OutputType> tc = new TestCase<InputType,OutputType>();
        tc.input=v;
        tc.setFeature(TestDataFeature.original);
    }
}

```



```

        this.addTestCase(tc);
        return tc.id;
    }

    /**
     * Add a test case into the test pool
     * @param tc: the test case to be added
     */
    public void addTestCase(TestCase tc) {
        testSet.add(tc);
    }

    /**
     * Remove a test case from the test pool
     * @param tc: the test case
     */
    public void removeTestCase(TestCase tc) {
        for (TestCase x: testSet) {
            if (x.id == tc.id) { testSet.remove(x); }
        }
    }

    /**
     * remove one test case from the test pool
     * @param id: the identity of the test case;
     */
    public void removeTestCase(String id) {
        for (TestCase x: testSet) {
            if (x.id.equals(id)) { testSet.remove(x); }
        }
    }

    /**
     * remove a set of test cases from the test pool
     * @param tcList: the test cases to be removed
     */
    public void removeAllTestCases(List<TestCase> tcList) {
        testSet.removeAll(tcList);
    }

    /**
     * Set the output value of the test case
     * @param v: the value of the output of the test case
     * @param tcIndex: the id number of the test case
     */
    public void setOutput(OutputType v, String id) {
        for (TestCase x: testSet) {
            if (x.id.equals(id)) { x.output = v; }
        }
    }

    /**
     * Get the test case of a given id number
     * @param index: id number of the test case
     * @return: the test case of the id number in the test pool
     */
    public TestCase<InputType,OutputType> get(String id){
        for (TestCase tx : testSet) {
            if (tx.id.equals(id)) {
                return tx;
            }
        }
        return null;
    }

    /**
     * Convert the test pool into a string
     */
    public String toString() {
        String outputStr = "[";
        for (TestCase x : testSet) {
            outputStr += x.toString() + ",\n";
        }
    }

```

```
        if (outputStr.length()>2) {  
            outputStr = outputStr.substring(0, outputStr.length()-2) + "];"  
        }else {  
            outputStr = outputStr + "];"  
        };  
        return outputStr;  
    }  
}
```

Appendix B. Example of Morphy Test Specification

The Java code below is an example of Morphy test specification for testing the trigonometric function $\sin(x)$.

```
package morphy.examples;
import morphy.annotations.*;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import javax.swing.JOptionPane;

import morphy.annotations.Datamorphism;
import morphy.annotations.MakeSeed;
import morphy.annotations.Metamorphism;
import morphy.annotations.TestCase;
import morphy.annotations.TestCaseFilter;
import morphy.annotations.TestCaseMetric;
import morphy.annotations.TestExecutor;
import morphy.annotations.TestPool;
import morphy.annotations.TestSetContainer;
import morphy.annotations.TestSetFilter;
import morphy.annotations.TestSetMetric;

public class SinTest {

    @TestSetContainer(
        inputTypeName = "Double",
        outputTypeName = "Double"
    )
    public TestPool<Double, Double> testSuite = new TestPool<Double, Double>();

    @TestExecutor
    public Double testSinX(Double x) {
        return Math.sin(x);
    }

    @MakeSeed
    public void randomValues(){
        Random randomGenerator = new Random();
        for (int i=0; i<100;i++){
            TestCase<Double, Double> tc = new TestCase<Double,Double>();
            tc.input = randomGenerator.nextDouble()*Math.PI/2;
            tc.feature = TestDataFeature.original;
            tc.setType("randomValues");
            testSuite.addTestCase(tc);
        }
    };

    @Datamorphism
    public TestCase<Double,Double> piMinus(TestCase<Double,Double> seed){
        TestCase<Double,Double> mutant = new TestCase<Double,Double>();
        mutant.input = Math.PI - seed.input;
        return mutant;
    }

    @Datamorphism(filter = "tooClose")
    public TestCase<Double,Double> mid(TestCase<Double,Double> x1, TestCase<Double,Double> x2) {
        TestCase<Double,Double> mutant = new TestCase<Double,Double>();
        mutant.input = (x1.input + x2.input)/2;
        return mutant;
    }

    @TestSetMetric
    public double avgDistance() {
        double total = 0;
        double num = 0;
        for (TestCase<Double, Double> x: testSuite.testSet ) {
            total += x.input;
            num++;
        }
    }
}
```

```

    };
    if (num==0) {return 0;}
    else {return total / num ;}
}

@TestSetFilter
public void sparse() {
    List<TestCase> toBeRemovedTCs = new ArrayList<TestCase>();
    int testSetSize = testSuite.testSet.size();
    for (int i=0; i<testSetSize-1; i++) {
        TestCase<Double, Double> x = testSuite.testSet.get(i);
        Double distanceX = null;
        for (int j=i+1; j<testSetSize; j++) {
            TestCase<Double, Double> y = testSuite.testSet.get(j);
            double distanceXY = Math.abs(x.input - y.input);
            if (distanceX == null) {
                distanceX = distanceXY;
            }else {
                if (distanceX > distanceXY) {
                    distanceX = distanceXY;
                }
            }
        }
        if (distanceX !=null && distanceX<0.00001) {
            toBeRemovedTCs.add(x);
        }
    }
    testSuite.removeAllTestCases(toBeRemovedTCs);
}

@TestCaseMetric
public double distance(TestCase<Double,Double> x) {
    Double distance = null;
    for (TestCase<Double,Double> y: testSuite.testSet) {
        if (x.id.equals(y.id)) {continue;}
        double distanceXY = Math.abs(y.input - x.input);
        if (distance == null) {distance = distanceXY;
        }else {
            if (distanceXY < distance) { distance = distanceXY; }
        }
    }
    return distance;
}

@TestCaseFilter
public Boolean tooClose(TestCase<Double,Double> x) {
    return (distance(x) <0.00001);
}

@Metamorphism(
    applicableTestCase="mutant",
    applicableDatamorphism = "piMinus",
    message="Failed the rule: Sin(x) = Sin(pi - x)")
public boolean PiMinusAssertion(TestCase<Double, Double> tc) {
    TestCase<Double, Double> originalTc = testSuite.get(tc.getOrigins().get(0));
    return (Math.abs(tc.output - originalTc.output) <= 0.0000000000000001);
}

@Analyser
public void visualization() {
    new VisualizeOutput(testSuite.testSet);
}
}

```

Appendix C. Template of Morphy Test Specification

```
//Datamorphic Test Specification
import morphy.annotations.*;
public class newTestSpec{
    @TestSetContainer(
        inputTypeName = "inputTypeName",
        outputTypeName = "outputTypeName")
    public TestPool<inputType, outputType> testSuite = new TestPool<inputType, outputType>();
    //TODO: Replace 'inputType' and 'outputType' with the actual datatype names
    //TODO: you can also change the name of testSuite

    @MakeSeed
    public void GenerateTestCases1(){
        //TODO use 'testSuite.addInput(<inputValue>)' to add an input value to the test case
    }

    @Datamorphism(
        filter = "a test case filter name")
    public TestCase<inputType, outputType> DM1 (TestCase<inputType,outputType> seed){
        TestCase<inputType, outputType> mutant = new TestCase<inputType, outputType>();
        //TODO write code to generate a mutant test case
        return mutant ;
    }

    @Datamorphism(
        filter = "a test case filter name" )
    public TestCase<inputType, outputType> DM2 (TestCase<inputType,outputType> seed1,
        TestCase<inputType, outputType> seed2){
        TestCase<inputType, outputType> mutant = new TestCase<inputType, outputType>();
        //TODO write code to generate a crossover test case
        return mutant;
    }

    @TestExecutor
    public outputType execute(inputType x) {
        outputType y = new outputType();
        //TODO: write the code M in the above line that invokes the method/program under test
        return y;
    }

    @Metamorphism(
        applicableTestCase= "mutant", // can als be "seed" or "all"
        //TODO: choose one of the above options.
        applicableDatamorphism = "datamorphism name",
        //TODO: insert the datamorphism name to the above
        message="Failed the metamorphic rule: ... ")
    //TODO: Change the above error message to be shown when the metamorphism returns false on a test
    public boolean MM1(TestCase<inputType, outputType> tc) {
        TestCase<inputType, outputType> originalTc = testSuite.get(tc.origins.get(0));
        boolean assertion = true;
        //TODO: write the code of assign a boolean value to the variable assertion
        return assertion;
    }

    @TestSetMetric
    public double testSetMetric1() {
        double metricValue = 0;
        //TODO: write the code to measure the quality of the test set stored in the test pool
        return metricValue;
    }

    @TestSetFilter
    public void testSetFilter1() {
        //TODO: write the code to reduce the test set in the test pool by removing low quality test cases
    }

    @TestCaseMetric
    public double testCaseMetric1(TestCase<inputType, outputType> tc) {
        double metricValue = 0;
        //TODO: write the code to measure the quality of the test case in context of the test pool
        return metricValue;
    }
}
```

```

    }

    @TestCaseFilter
    public Boolean testCaseFilter1(TestCase<inputType, outputType> tc) {
        boolean toBeRemoved = true;
        //TODO: write the code to decide whether the test case is to be removed
        return toBeRemoved;
    }

    @Analyser
    public void analyser1() {
        boolean toBeRemoved = true;
        //TODO: write the code to analyse the test set
    }
}

```