

Automation of Datamorphic Testing

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: dmliu@njust.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. This paper proposes a set of test strategies that combine datamorphisms to generate test sets that adequately cover various types of mutant test cases. These strategies are formally defined. Their implementation algorithms are provided. The correctness of the algorithms are proved. The paper also illustrates their uses for testing both traditional software and AI applications with three case studies.

Keywords-Software test, Test automation, Artificial Intelligence, Test tools, Datamorphic test

I. 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 [9]. 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 [3], [18], [29], [30] have demonstrated that it is 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 the test code of datamorphisms, metamorphisms, seed makers, etc, to be reused even across

different projects.

- 2) *Composability* of test code in different combinations to conduct different experiments with the software under test.
- 3) *Constructability* of the 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 test tool called *Morphy*¹.

The paper is organised as follows. Section II extends the datamorphic testing framework. Section III presents the *Morphy* test tool. Section IV defines a set of strategies that combines datamorphisms to generate test data. Section V reports three case studies to demonstrate the uses of *Morphy*. Section VI concludes the paper by a comparison with related work and a discussion of future work.

II. 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 and 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.

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

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

A. 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:

{metamorphismName : (pass|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 [31].

B. 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 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 executers* 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.

C. Test Specifications

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 [31] for an example. Each test morphism is annotated with metadata to declare the type of test morphism that the method belongs to. Table I lists the annotations and datatypes of various types of test morphisms as implemented in Morphy.

Table I
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

III. TEST TOOL MORPHY

As shown in Figure 1, Morphy consists of three main facilities: *test management*, *test runner* and *test scripting*. The test management facility enables test sets to be saved into files, loaded from files and edited in a graphic user interface. It also enables a test specification to be loaded into the system so that various test morphisms of the test

specification can be executed by the test runner. The test runner also implements various test strategies that combine test morphisms to achieve advanced test automation. The test scripting facility enables interactive testing activities to be recorded as test scripts, saved into files, reloaded from files and replayed.

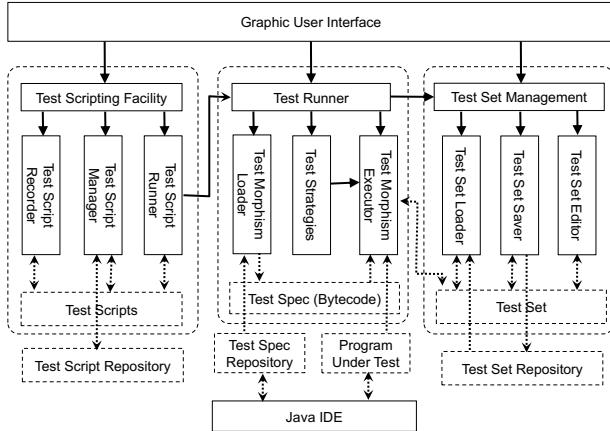


Figure 1. The Architecture of Morphy

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 of test specification.

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 for the management of test entities, performing test activities by invoking various types of test morphisms, applying test strategies, and recording-replaying test scripts, respectively.

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 as input to perform the interactive and automatic testing functions.

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

Morphy enables test automation at three different levels. At the lowest level, various test activities can be performed by invoking test morphisms via a click of buttons on Morphy's GUI. 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. It is particularly useful for repeated testing processes, such

as in regression testing and repeated experiments with the software under test to obtain data for statistical analysis.

IV. MUTANT COMBINATION STRATEGIES

Test strategies play a crucial role in the automation of datamorphic testing because they can combine and compose test morphisms together effectively to achieve commonly occurring software testing requirements. This section proposes a set of test strategies that combines datamorphisms to enrich a given test set with adequate coverage of different types of mutant test cases.

A. First Order Mutant Coverage

A datamorphic approach to testing AI applications uses seed test cases to test the normal operation condition of the AI system under test, and uses datamorphisms to transform a test case that represents other operation conditions that can be derived from the normal operation conditions [29].

For example, to test face recognition applications, datamorphisms are developed to transform the images of human faces by editing the facial attributes, such as adding makeup, wearing glasses, changing skin tones, change hair styles and colour, etc. In Figure 3, (a) is the original photo, (b), (c), and (d) are images obtained by applying datamorphisms of adding spectacles, applying makeup, and changing hair styles, respectively. They are used to test face recognition applications [29].

Similarly, in [25], for testing driverless vehicles, datamorphisms are developed to alter the weather condition of a recorded driving process to be in fog, to transform the lighting condition from daytime to nighttime with street lights, etc. Such transformed test cases are called the mutant test cases, and can formally be defined as follows.

Let T be the set of all possible test cases for the software under test, $S \subset T$ ($S \neq \emptyset$) be a set of test cases, and $D \neq \emptyset$ be a set of datamorphisms and $d \in D$ be 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

In the context of testing AI applications where datamorphisms are transformations of test cases that alter the key features of the test cases to represent different operation conditions, it is important for cover each operation condition adequately. Thus, here we propose a test adequacy criterion called *first order mutant completeness*.

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

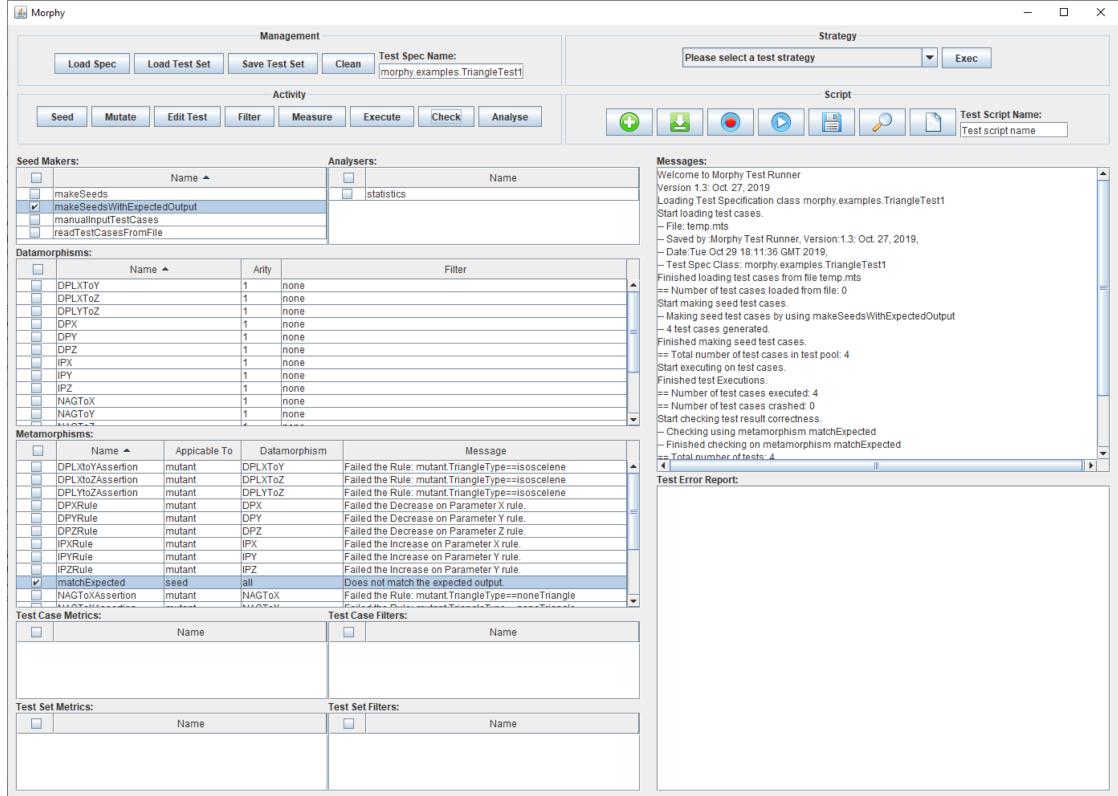


Figure 2. Morphy's Main GUI

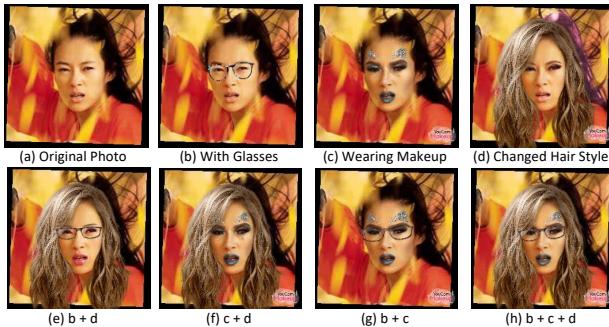


Figure 3. Mutants for Face Recognition

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 given 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
  C = EmptySet;
  for (each datamorphism d in D) {
    tempT = EmptySet;
    Assume that d is a k-ary datamorphism;
    forall k-tuples  $(x_1, \dots, x_k)$  of  $S$  {
      add  $d(x_1, \dots, x_k)$  to tempT;
    };
    C = C + tempT;
  };
  return C + S;
End

```

The following theorem asserts the correctness of the algorithm. The proof can be found in [31].

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 . \square

B. Higher Order Mutant Coverage

Datamorphisms can also be applied to test cases multiple times to generate mutants of mutants, which are called *high order mutants* and formally defined as follows. 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

For many AI applications, higher order mutants are important test cases. For example, to test a face recognition application, it is desirable to include test cases that are obtained by adding to the image of a human face a pair of glasses, applying makeup and dying the hair colour. In Figure 3, image (e)-(g) are test cases of 2nd order mutants, while image (h) is a 3rd order mutant of the original image. In testing a driveless car, it is desirable to have test cases representing an operating condition where the road condition is wet and weather is foggy, and lighting is night time. Such operating conditions may occur at the same time. Therefore, it is necessary to test the system on such a combination of operating conditions.

Similar to first order mutant completeness, a test set is second order mutant complete if it contains all seed test cases, all first order mutants and all second order mutants. In general, we have the following test adequate criterion.

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

The following can be proved based on Theorem 1 by induction on the order K .

Corollary of Theorem 1: By repeating Algorithm 1 for K times such 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. \square

C. Datamorphic Combination Coverage

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 given 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 cost of testing can 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 4 below shows some examples of mutants,

in which (a) and (b) are first order mutants, and (c) to (f) are second order mutants.

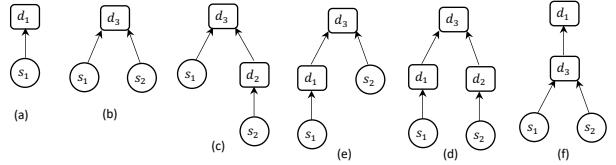


Figure 4. Examples of Mutant Trees

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 substituting 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: (Combinatorial Coverage)

A set \mathcal{C} of datamorphism combinations is *combinatorial complete* for D , if for all non-empty subsets $D' \subseteq D$, there is a combination $c \in \mathcal{C}$ such that D' is the set of datamorphisms in c .

A set C of test cases is *combinatorial complete* with respect to S and D , if

- there is a set \mathcal{C} of datamorphism combinations that is combinatorial complete with respect to D ; and
- for every combination $c \in \mathcal{C}$, if c is k -ary, then for all k -tuples of test cases $(x_1, \dots, x_k) \in S^k$, there is a test case y in C 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
    for (each datamorphism d in D) {
        tempT = empty_set;
        Assume d is a k-ary, where k>0;
        for (all k-tuples (x1, ..., xk) of S) {
            add d(x1, ..., xk) to tempT;
        };
        S = S + tempT;
    };

```

```

    return C + S;
End

```

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

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

V. CASE STUDIES

We have conducted three case studies on the development of Morphy test specifications and the uses of Morphy in automated software testing. The purpose of the case studies is to examine the usability of the proposed test automation approach in general and the testing tool Morphy in particular. We focus on the following features.

- 1) the *feasibility* of the proposed software test automation approach: how much effort is required to develop test morphisms to achieve test automation for various different types of software systems including AI applications.
- 2) the *composability* of test morphisms and test strategies to form automated test processes for different types of software systems.
- 3) the *reusability* of the test morphisms: whether the test morphisms developed for testing one software system can be reused for testing a different system.

These case studies are summarised below.² More details can be found in [31].

A. Triangle Classification

Triangle classification is a classic software testing problem that Myer used to illustrate the importance of combining various types of test cases [19]. 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.*” [19] Myer listed 14 questions for testers to assess the adequacy of a test and reported that, for such a seemingly simple program, “*highly qualified professional programmers score, on the average, only 7.8 out of a possible 14*”.

The case study demonstrated that datamorphisms can easily be developed to achieve test adequacy and the testing process can be automated. A set of 20 datamorphisms was developed, inspired by Myer’s test criteria. When the first order mutant complete strategy is applied to these datamorphisms on 4 seed test cases, 80 mutant test cases are generated automatically; these fully meet Myer’s test criteria. Moreover, for each datamorphism, we also developed a corresponding metamorphism to check the correctness of the program under test. Four different programs were tested: two incorrect ones and two correct ones using different

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

algorithms. The testing successfully detected the bugs in the faulty programs, while the correct ones passed the test. The test specifications were split into two classes, so the test morphisms were reused; see Figure 5.

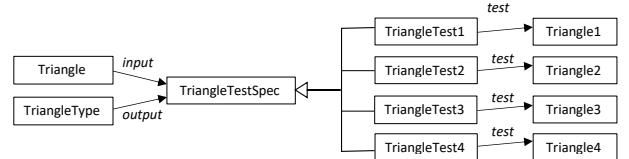


Figure 5. Structure of Test Specification

B. Trigonometric Functions

Three trigonometric functions $\sin(x)$, $\cos(x)$ and $\tan(x)$ provided by Java’s Math class are tested. The correctness of the library’s implementation of these functions is checked against a set of 27 trigonometric identities implemented as metamorphisms; see Table II.

Two seed makers were written: one generates a set of 17 special values between 0 and 2π ; the other generates 100 random test cases in the range between 0 and $\frac{\pi}{2}$.

Unary datamorphisms were written for the mappings from x to $2\pi \pm x$, $\pi \pm x$, $\frac{\pi}{2} \pm x$, $-x$, and binary datamorphisms from x and y to $x \pm y$, and $\frac{x+y}{2}$.

The first order mutant complete and the combination complete test sets were generated using Morphy testing tool. The testing revealed an error rate of 0.957%, which are on test cases where the inputs to $\tan(x)$ are not defined, or very close to undefined.

Table II
LIST OF METAMORPHISMS

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

C. Face Recognition

The experiments with face recognition reported in [29], [30] are repeated, but the test code is re-written in the form of Morphy test specifications. The case study clearly demonstrated the benefit of test automation and the reusability of

test code achieved by Morphy. The test data for testing a face recognition application are images of sizes over 100 KB. In [18], [29], [30], 200 images of different persons were used and each generated 13 mutants using AttGAN [12] to alter the facial features. In the case study, each mutant image was only generated once and stored in the file system, then it was reused to test different face recognition applications rather than generated many times.

In addition to repeating the previous experiments, which examines whether a face recognition application recognises a person from a mutant image, a new experiment was designed to examine whether a face recognition application rejects a mutant images of a different person. The new experiment is implemented by writing just one new seed maker. All other test morphisms are the same as the existing ones. The test specifications are split into three classes: one for datamorphisms and analysers, one for seed makers and one for test executors. This is in the similar structure to the test specifications for Triangle Classification.

Test scripts were recorded and slightly edited to add code for repeating tests a number of times in order to obtain statistically significant data. A test analyser method was also written to do statistical analysis of the experiment data. The test process was highly automated and repeatable.

D. Results of The Case Studies

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

Table III
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

Second, test specifications are reusable especially when they are properly structured. In the case study, test specifications are decomposed into a number of classes where common test morphisms are placed together. They are inherited by classes that contain test specific morphisms.

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

VI. CONCLUSION

A. Main Contributions

The main contributions of the paper are as follows.

First, this paper redefines the method of datamorphic testing method by classifying test artefacts into test entities and morphisms. Datamorphisms, metamorphisms and seed makers are examples of test morphisms. We have also identified a set of other test morphisms, which include test case metrics and filters, test set metrics and filters, test executors and analysers. The case studies reported in this paper have clearly demonstrated the importance of the test morphisms of test executors and analysers. The other types of test morphisms also play crucial roles in the implementation of test strategies; this will be reported in separate papers.

Second, the paper proposes a novel framework of test automation and demonstrates its feasibility by a test automation tool called *Morphy*. In this framework, basic testing activities can be automated by writing test codes for various test morphisms and invoking them through a test tool like Morphy. Advanced combinations of test morphisms can be realised by test strategies to achieve a higher level of test automation. Test automation can be further improved by a test scripting facility through recording the interactive invocations of test morphisms and test strategies as well as test management activities such as loading test specifications, loading and saving test sets, etc.

Third, this paper proposed a set of test strategies for combining datamorphisms to achieve adequate coverage of mutant test cases. These strategies are particularly useful for testing AI applications such as face recognition and driveless vehicles. They are formally defined and their implementation algorithms are presented and their correctness are formally proved. As far as we know, this is the first time test strategies are formally studied in the literature of software testing.

Finally, the paper reports three case studies with the datamorphic testing tool Morphy. They clearly demonstrated that the proposed approach to test automation in general and the Morphy tool in particular have the benefit that test automation code represented in the form of test morphisms are highly reusable and composable to construct test automation processes flexibly and easily. It is applicable to all kinds of software systems including AI applications.

B. Related Work

There are two kinds of test automation frameworks: XUnit [10], [17] like JUnit and GUI based test automation tools like Selenium [22] and WebDriver [26]. In comparison with them, Morphy provides more advanced test automation facilities such as test strategies.

In XUnit framework, a test 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 specifications are declaratively imperative in the sense that each test class declares various testing morphisms 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.

GUI based test automation tools employ test scripts or test code to interact with GUI elements. The most representative and most well-known example of such testing tools is Selenium [22]. It has two test environments: (a) the Selenium IDE in which manual testing can be recorded into test scripts and replayed; (b) the Web Drivers, which provided an API for writing test code in programming languages. Morphy also employs test scripts, but it is equipped with more advanced test automation facilities such as test strategies, so it achieves a higher level of test automation.

An advantage of Morphy is that the architecture enables various testing techniques and tools to be integrated by wrapping existing testing tools as methods in a test specification to invoke the tools. For example, test case generation techniques and tools [1] like fuzz testing [24], data mutation testing [21], random testing [2], adaptive random testing [7], [16], combinatorial testing [20] and model based test case generators are all test morphisms, which can be wrapped as seed makers or datamorphisms. Metamorphic relations in metamorphic testing [6] and formal specification-based test oracles [4], [5], [14], [15], [28] are metamorphisms. Test coverage measurement tools like [23] 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 [8], [11] can be regarded as test strategies. Therefore, they can all be easily integrated into Morphy.

C. Future Work

The test adequacy criteria of 1st order mutant coverage, higher order mutant coverage, and combinatorial mutant coverage are proposed in this paper and are based on our experiences in testing AI applications. Assessing their effectiveness for AI applications is on our agenda for future work.

In addition to the strategies defined in this paper, Morphy has already implemented two other types of test strategies: (a) exploration strategies, which explore the test space in order to find the borders between subdomains for testing the classification and clustering type of AI applications; (b) test set optimisation strategies, which employ genetic algorithms to optimise test sets. They will be reported in separate papers. The case studies reported in this paper have used test scripts intensively to improve test automation. A

more detailed study of the test script facility will be reported in a separate paper, too.

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 [13] and fault injection tools for fault-based testing methods are test morphisms that are mappings from programs to programs or to sets of programs. Specification mutation operators are test morphisms that map from formal specifications to specifications, or to sets of specifications. It is an interesting further research question to ask how to integrate such test morphisms into the datamorphic testing tools like Morphy, although, theoretically speaking, there should be no significant difficulty in doing so.

It is also possible to integrate XUnit like JUnit and GUI based test automation tools like WebDriver with Morphy. It will be interesting to see how the informal guidelines of exploration testing strategies can be formalised and implemented in the datamorphic testing framework.

REFERENCES

- [1] S. Anand, *et al.* An orchestrated survey of methodologies for automated software test case generation. *JSS*, 86(8):1978 – 2001, 2013.
- [2] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE TSE*, 38(2):258–277, March 2012.
- [3] Y. Bagge. Experiments with testing a bounded model checker for C. MSc Dissertation, School of Eng., Comp. and Math., Oxford Brookes Univ., Oxford, UK, Sept. 2019.
- [4] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *SEJ*, pp387–405, Nov. 1991.
- [5] H. Chen, T. Tse, and T. Chen. Tackle: a methodology for object-oriented software testing at the class and cluster levels. *ACM TOSEM*, 10(1):56–109, 2001.
- [6] T. Y. Chen, *et al.* Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, Jan. 2018.
- [7] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proc. of ASCC 2004.*, LNCS 3321, pp320–329. Springer, 2004.
- [8] M. Dave and R. Agrawal. Search based techniques and mutation analysis in automatic test case generation: A survey. In *Proc. of ACC 2015*, pp795–799, June 2015.
- [9] A. Gotlieb, M. Roper, and P. Zhang, eds. *Proc. of The 1st IEEE Int'l Conf. on Artificial Intelligence Testing (AITest 2019)*. IEEE Computer Society, Apr 2019.
- [10] P. Hamill. *Unit Test Frameworks*. O'Reilly, 2005.

[11] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):Article 11, 61 pages, Nov. 2012.

[12] Z. He, *et al.* AttGAN: Facial attribute editing by only changing what you want. *IEEE Trans. on Image Processing*, 28(11):5464–5478, Nov 2019.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, Sep. 2011.

[14] L. Kong, H. Zhu, and B. Zhou. Automated testing EJB components based on algebraic specifications. In *Proc. of COMPSAC 2007(2)*, pp717–722, 2007.

[15] D. Liu, X. Wu, X. Zhang, H. Zhu, and I. Bayley. Monic testing of web services based on algebraic specifications. In *Proc. of SOSE 2016*, Oxford, UK, March 2016.

[16] Y. Liu and H. Zhu. An experimental evaluation of the reliability of adaptive random testing methods. In *Proc. of SSIRI 2008*, pp24–31, Yokohama, Japan, July 2008.

[17] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[18] S. Mugutdinov. Applying datamorphic technique to test face recognition applications. BSc Dissertation, Oxford Brookes Univ., Oxford, UK, Mar. 2019.

[19] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, Inc., 2nd ed., 2004.

[20] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, Feb. 2011.

[21] L. Shan and H. Zhu. Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. *Computer Journal*, 52(5):571–588, Aug 2009.

[22] Software Freedom Conservancy. Selenium website. Online at URL: <https://selenium.dev>, Nov. 2019.

[23] Stackify. The ultimate list of code coverage tools. Online at URL: <https://stackify.com/code-coverage-tools/>, May 2017.

[24] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[25] Y. Tian, K. Pei, S. Jana, and B. Ray, DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. in *Proc. of ICSE 2018*, pp. 303–314, May 2018.

[26] WWW Consortium. WebDriver: W3C Recommendation. At URL: <https://www.w3.org/TR/webdriver1/>, June 2018.

[27] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *JSTVR*, 22(2):67–120, Mar. 2012.

[28] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *Proc. of QSIC'03*, pp91–99, Dallas, USA, Nov. 2003.

[29] H. Zhu, *et al.* Datamorphic testing: A method for testing intelligent applications. In *Proc. of AITest 2019*, pp149–156, Apr 2019.

[30] H. Zhu, *et al.* Datamorphic testing: A methodology for testing AI applications. Technical Report OBU-ECM-AFM-2018-02, School of Eng., Comp. & Math., Oxford Brookes Univ., Oxford, UK, Dec 2018. Available at ArXive <http://arxiv.org/abs/1912.04900>

[31] H. Zhu, *et al.* Morphy: A datamorphic software test automation tool. Technical Report OBU-ECM-AFM-2019-01, School of Eng., Comp. and Math., Oxford Brookes Univ., Oxford, UK, Dec. 2019. Available at ArXive <http://arxiv.org/abs/1912.09881>.