Datamorphic Testing: A Method for Testing Intelligent Applications

Hong Zhu*, Dongmei Liu†, Ian Bayley*, Rachel Harrison*, Fabio Cuzzolin*
*School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK
Email: (hzhu, ibayley, rachel.harrison, fabio.cuzzolin)@brookes.ac.uk
†School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China
Email: dmliukz@njust.edu.cn

***Abstract*—Adequate testing of AI applications is essential to ensure their quality. However, it is often prohibitively difficult to generate realistic test cases or to check software correctness. This paper proposes a new method called datamorphic testing, which consists of three components: a set of seed test cases, a set of datamorphisms for transforming test cases, and a set of metamorphisms for checking test results. With an example of face recognition application, the paper demonstrates how to develop datamorphic test frameworks, and illustrates how to perform testing in various strategies, and validates the approach using an experiment with four real industrial applications of face recognition.**

***Index Terms*—software testing, intelligent applications, machine learning, test case generation, test oracle, datamorphism, metamorphism.**

I. Motivation

We have seen a rapid growth in the application of machine learning (ML), data mining and other artificial intelligence (AI) techniques in recent years. Typical examples of such applications include driverless vehicles, face recognition and finger print recognition in security control, workload pattern learning in computer cluster operation, personalisation of social media networking for business intelligence, situation recognition and action rule learning in healthcare, smart homes and smart cities, etc. Many such applications are also closely integrated with robotics, the Internet-of-Things, Big Data, and Edge, Fog and Cloud computing. These all require automated and optimised data collection and processing. AI techniques, especially ML, have been widely regarded as a promising solution to the underlying hard computational problems. All such applications must be thoroughly tested to ensure their quality [1].

However, the current practice of testing AI applications lags far behind the maturity of testing traditional software. Testers have been confronted with grave challenges because the distinctive features of ML applications disqualify existing software testing methods, techniques and tools. As a result, generating realistic test cases and checking the results are prohibitively difficult and expensive [2].

To address these difficulties, we propose a novel approach called datamorphic testing for AI applications based on our previous work [3] on integration of data mutation testing [4] and metamorphic testing [5], [6].

The paper is organized as follows. Section II introduces the basic concepts of datamorphic testing method. Section III discusses the process and various strategies of datamorphic testing. Section IV reports an experiment with four real industrial ML applications of face recognition. Section V discusses related work and future work.

II. Basic Concepts of Datamorphic Testing

We first define the key concepts underlying the proposed testing method and illustrate them using face recognition as an example. Here, facial recognition is the problem of determining whether an image portrays somebody whose facial image is stored in a database of known persons.

A. Datamorphism

A *datamorphism* is a transformation that derives new test data, called *mutants*, from existing test data. Let D and C be the input and output domains of a program P under test, respectively, $V(x_1, x_2, \cdots, x_k, l)$ be a predicate on the set $D^k \times L$, where L is a given set of parameters, and $k \geq 0$ is a given natural number.

Definition 1: (Datamorphism)

A k-ary *datamorphism* φ is a mapping from $D^k \times L$ to D such that for all $\overrightarrow{x} = (x_1, x_2, \cdots, x_k) \in D^k$, $l \in L$, if $V(\overrightarrow{x}, l) = true$, we have that $\varphi(\overrightarrow{x}, l) \in D$. The elements l in set L are called the *parameters* of the datamorphism. V is called *applicability condition* of the datamorphism. □

For example, for the face recognition application, the input domain of the application contains images of human faces. The following examples of datamorphisms are applicable to human facial images:

1) Add a pair of glasses;
2) Add makeup;
3) Change hair style;
4) Change hair colour.

Fig. 1 shows the results of applying these datamorphisms to photos; (a) is the original photo[1], (b) adds a pair of glasses to (a), (c) adds a pair of sunglasses, (e) to (g) add makeup, (h) changes the hairstyle and colour. Images (i) and (j) are obtained by transforming (a) into black-and-white and into watercolours, respectively.

[1]From the public dataset at URL http://vis-www.cs.umass.edu/lfw/

Note that, first, some transformations are not meaningful in certain contexts. For example, automated passport control requires people to remove items that obscure the face so adding glasses is inapplicable there.

Second, most datamorphisms could be implemented as program components. For example, images (b) to (h) in Fig. 1 were obtained by using a mobile phone app called YouCam[2] and images (i) and (j) were obtained using another mobile phone app called Prisma[3]. In the experiment reported in Section IV, we also use the facial attribute inverting operators provided by the AttGAN system [7].

B. Metamorphism

Not only do datamorphisms provide a means of test case generation, they are also a powerful means to specify the required functions of the application in terms of relationships between test cases and the expected outputs. Such a relationship is called a metamorphic relation in the literature of software testing [5], [6].

Definition 2: (Metamorphic Relation)

Let $k \geq 1$ be a natural number. A k-ary metamorphic relation M for program P is a relation on $D^k \times C^k$ such that program P is correct on input $\overrightarrow{x} = (x_1, \cdots, x_K) \in D^k$ implies that the relation $M(\overrightarrow{x}, P(x_1), \cdots, P(x_k))$ holds, where $P(x)$ is program P's output on input x. $\square$

The following is a typical example of metamorphic relation for the $Sin(x)$ function, where x and y can be any real number.

$$(x + y = \pi) \Rightarrow Sin(x) = Sin(y)$$

The above equation defines a binary metamorphic relation (i.e. $k = 2$) for the Sin function.

A typical form of metamorphic relations is

$$V(x_1, \cdots, x_n) \Rightarrow R(x_1, \cdots, x_n, y_1, \cdots, y_n),$$

where $V(x_1, \cdots, x_n)$ is a relation on the input data $x_1, \cdots, x_n$, $y_1, \cdots, y_n$ are the corresponding outputs, and $R(x_1, \cdots, x_n, y_1, \cdots, y_n)$ is a relation on them. It asserts that for all input data $x_1, \cdots, x_n$ satisfying condition V, the corresponding outputs $y_1, \cdots, y_n$ from program P must satisfy condition R. The condition V is called the *applicability* condition. In practice, an applicability condition may well be constructed and defined in terms of the output, even the immediate results, of the program P on the inputs $x_1, \cdots, x_n$. Condition R is called the *correctness* condition. It is an assertion about software correctness in terms of an expected relationship between inputs and outputs.

A metamorphic relation is a very flexible and expressive means of specifying software functions. In fact, formal specifications in the form of pre/post-conditions $pre(x)\{P\}post(x, y)$ is a special case of metamorphic relation $pre(x) \Rightarrow post(x, P(x))$, where the arity of the relation $k = 1$.

To apply a metamorphic relation in the form of $V(\overrightarrow{x}) \Rightarrow R(\overrightarrow{x}, \overrightarrow{y})$ in software testing, one must generate test cases $a_1, \cdots, a_n$ that satisfy condition $V(a_1, \cdots, a_n)$, for example, by searching on the input space or by solving constraints. Then, the program P under test is executed on test cases $a_1, \cdots, a_n$ to obtain outputs $b_1 = P(a_1), \cdots, b_n = P(a_n)$. Finally, the condition R is checked on the input and outputs $a_1, \cdots, a_n, b_1, \cdots, b_n$ to determine whether the program is correct on these test cases.

The two main difficulties of using metamorphic relations in software testing are finding a suitable set of metamorphic relations that are effective for detecting faults in the software under test and finding test cases that satisfy the applicability conditions of the metamorphic relations. These difficulties can be eased if metamorphic relations are combined with datamorphisms as shown in [3].

Definition 3: (Metamorphism)

Let $\Psi \neq \emptyset$ be a given set of datamorphisms on the input domain of program P. A metamorphic relation M is called a *metamorphism*, if it can be presented in the following form.

$$R(x_1, \cdots, x_k, P(x_1), \cdots, P(x_k), P(x'_1), \cdots, P(x'_m))$$

where $x'_i = \varphi_i(\overrightarrow{z_i}, l_i)$, $\overrightarrow{z_i}$ is a subset of $\{x_1, \cdots, x_k\}$, $\varphi_i \in \Psi$ for all $i = 1, \cdots, m$. We say that the metamorphism is defined on $\varphi_1, \varphi_2, \cdots, \varphi_m$. $\square$

A metamorphism asserts that for all input data $x_1, \cdots, x_k$, the correctness condition R holds on $P(x_1), \cdots, P(x_k), P(x'_1), \cdots, P(x'_m)$. For the Sin function example the following is a metamorphism for the datamorphism $\phi(x) = \pi - x$.

$$Sin(x) = Sin(\pi - x)$$

With metamorphisms, testing can be performed automatically by, first, applying the datamorphism on existing test cases to obtain mutant test cases. Then, the program is executed on the seed and mutant test cases. Finally, the results of the test executions are checked against the correctness condition to determine the correctness of the program. This avoids searching for test cases or constraint solving.

In many cases, a metamorphism can be easily derived from the meaning of the datamorphism. For example, let $AddGlasses(x)$ denote the datamorphism of adding a pair of glasses on a facial image. A metamorphism for the face recognition application $FaceOf(x)$ can be formally defined on $AddGlasses(x)$ as follows.

$$FaceOf(x) = FaceOf(AddGlasses(x)).$$

This metamorphism states that if the face recognition application recognises a person in an image, then, after adding a pair of glasses by editing the image, the software should still recognise the person. Therefore, applying this metamorphism to image (a) and (b) in Fig. 1, we expect a face recognition application will identify that the images are of the same person.

Metamorphisms like the above are actually formal specifications. Testing an AI application can be automated if datamorphisms and metamorphisms can be implemented in

[2]URL: https://www.perfectcorp.com/app/ycp

[3]URL: https://prisma-ai.com

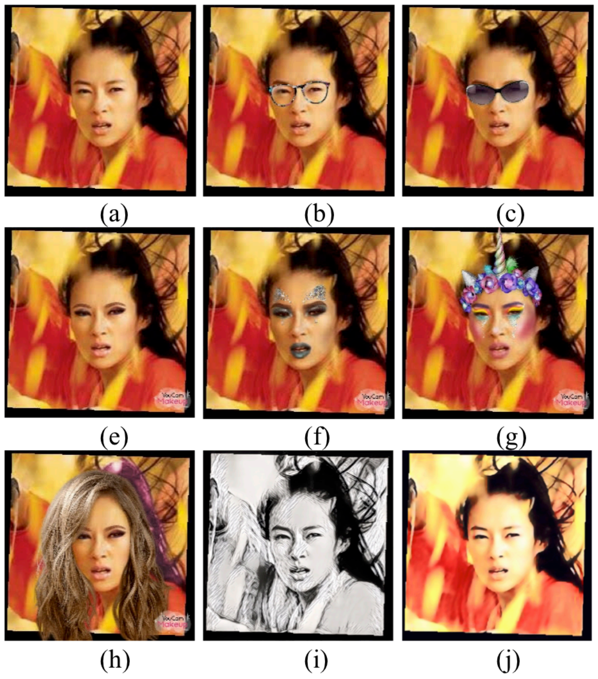


Fig. 1. Examples of datamorphisms on images.

software. For example, as stated earlier, it is possible to edit an image and generate a mutant of an image as shown in Fig. 1. Then, by feeding both the original and the mutant images to a facial recognition application, the correctness of the application can be checked by comparing the outputs of these two test cases. If the outputs are identical, the application passes the test; otherwise, an error is detected.

C. Seed Test Cases

For a datamorphism to be useful, we must have a set of known test cases, called the *seed test cases*, or simply *seeds*.

In the face recognition example, a seed could be an image of a person's face. Such a set of seeds is normally available for testers of many AI applications, for example, as training data for an ML application. The seeds could be a subset of such training data selected at random or according to certain criteria. However, seeds alone are inadequate. Our method uses the seeds to generate more test cases to make an adequate test of the application. Seeds are not necessarily labeled with the expected outputs unless the datamorphisms require such labels.

A datamorphism may well be applicable to mutants, especially when the mutants are generated by a different datamorphism. For example, in Fig. 1, (h) is obtained by applying a datamorphism on mutant (e).

In summary, our testing framework consists of three elements: a set of seed test cases, a set of datamorphisms and a set of metamorphisms.

Definition 4: (Datamorphic Test Framework)

Let D be the input domain of a program P under test. A datamorphic test framework $\mathfrak{F}$ is an ordered triple $\langle S, \Psi, \mathfrak{M} \rangle$, where $S \subseteq D$ is a finite subset of D. The elements of S are called the seed test cases, or simply seeds. Ψ is a finite set of datamorphisms, and $\mathfrak{M}$ is a finite set of metamorphisms. □

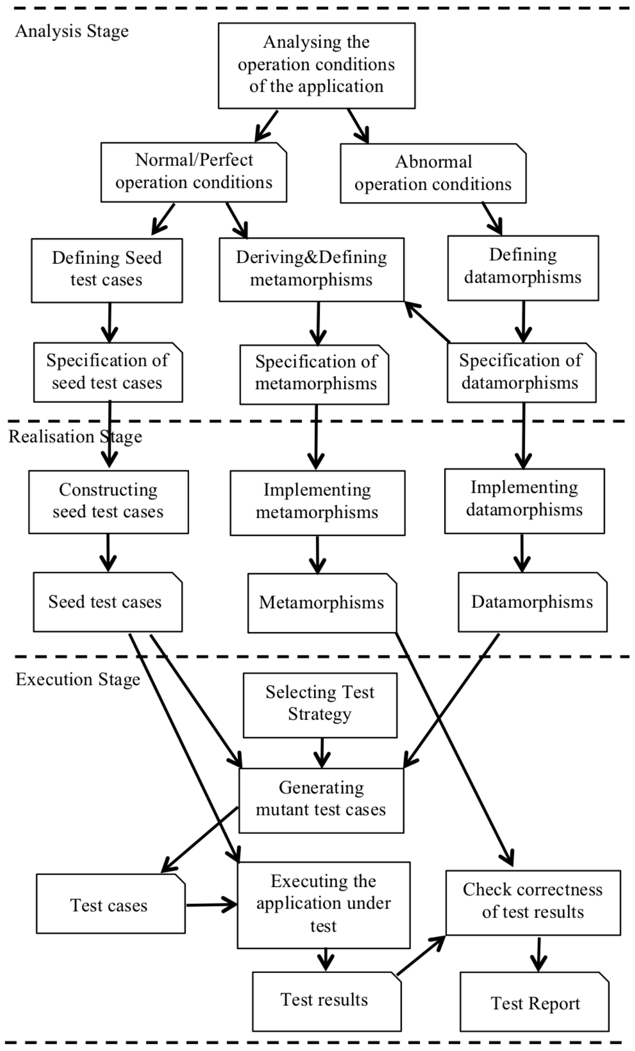

Analysis Stage
Analysing the operation conditions of the application
Normal/Perfect operation conditions
Abnormal operation conditions
Defining Seed test cases
Deriving&Defining metamorphisms
Defining datamorphisms
Specification of seed test cases
Specification of metamorphisms
Specification of datamorphisms
Realisation Stage
Constructing seed test cases
Implementing metamorphisms
Implementing datamorphisms
Seed test cases
Metamorphisms
Datamorphisms
Execution Stage
Selecting Test Strategy
Generating mutant test cases
Test cases
Executing the application under test
Check correctness of test results
Test results
Test Report

Fig. 2. Datamorphic testing process.

The next section discusses how to construct a datamorphic test framework and how such a test framework can be used with different strategies.

III. Testing Process And Strategies

A. Process of Datamorphic Testing

As illustrated in Fig. 2, the datamorphic testing process consists of three stages.

1) Stage 1: Analysis: The first stage is analysis of the testing problem in order to design a datamorphic test framework. In this stage, seed test cases, datamorphisms and metamorphisms are identified. These three elements are closely related to each other, thus should be engineered systematically.

Analysis starts by identifying the operating conditions of the application. For a face recognition application at an international airport's border control, for example, the input to the software is a photo from a camera fitted on an automatic passport checking machine and the photo of the passport holder retrieved from the information contained in a smart passport. The person may be of any ethnic background, age or gender. Normally, the person should be facing the camera directly without glasses and without heavy makeup, etc. However, in reality, people may wear glasses (even sunglasses), and have makeup. The photo could be many years old and taken from an unusual angle and the camera may have dust on its lens, etc. These form a variety of operating conditions of the application. Real world use of the software could be a combination of these aspects. Adequate testing of the application must cover all such combinations. Directly collecting test data to achieve adequate testing could be a challenge.

Our approach to solve this problem is to take a subset of the operating conditions as the normal conditions, and the others as abnormal conditions. The seed test cases are constructed from various combinations of the normal operating conditions. For example, we consider the person in front of the automatic passport checking machine is in normal operating condition if he/she does not wear glasses, has no makeup and the passport photo is taken recently. He/she could be of different ethnic background, in a different age group, and of different gender. Photos with combinations of these factors are sought as the seed tests.

The other operating conditions are regarded as "abnormal". For example, for the face recognition application to passport control, abnormal operating conditions include the situations that the passenger wears glasses or makeup, changes hair colour, the passport photo is out of date, the camera lens is dusty, or the camera points to the person at a different angle, etc. Test cases representing abnormal operation conditions are obtained by transforming seed test cases into mutant test cases. Each of these abnormal conditions in the operation of the software is therefore a candidate for datamorphism.

For an abnormal operating condition to be a datamorphism, it must also be feasible to implement a transformation on the input data. Otherwise, the operating condition must be added to the set of "normal" operating condition and the corresponding test cases must be obtained directly as seeds.

Once the normal and abnormal operating conditions are identified, the corresponding changes of a datamorphism on the output should be identified, thus metamorphisms can be derived. For example, adding a pair of glasses should make the person still recognizable.

The first stage should finish with a set of specifications of the seed test cases, the datamorphisms and the metamorphisms. These specifications can be in natural language, but should be detailed enough for performing the next steps of the process.

2) Stage 2: Realisation: The second stage of datamorphic testing is realisation. In this stage, the actual test data of the seeds are constructed, and datamorphisms and metamorphisms are realised.

A datamorphism can be realised by developing software that takes test data as input and generates new test data. Sometimes, applications already available can be used as datamorphisms as shown by the face recognition example. A datamorphism can also be realised by manually editing test cases. If datamorphisms cannot be realised, seed test cases must be obtained directly to represent the corresponding operating conditions.

A metamorphism can typically be implemented as code that invokes the application with seed and mutant test cases, and then stores and/or compares the result according to the metamorphism.

3) Stage 3: Execution: The final stage of datamorphic testing is execution, in which the test is executed according to a test strategy, which is discussed in detail in the next subsection.

B. Datamorphic Testing Strategies

There are many strategies of generating test cases using datamorphisms.

- Exhaustive

An exhaustive strategy is to generate all possible mutant test cases by repeatedly applying the datamorphisms on the seeds until no more new mutants can be generated. Formally, let T be the set of test cases. T is initialised to the set of seed test cases. Each datamorphism is applied to every seed with all possible parameters to generate a set of mutants. These mutants are added to the set T of test cases and duplicates are removed. This mutant generation process is repeated until no new test cases can be added.

Exhaustive testing may generate a huge number of test cases from a small set of seeds. In some cases, there may be an infinite number of test cases that can be generated from a finite number of seeds. Therefore, it is desirable to select a subset of such exhaustive test set. The following are some examples.

- Combinatorial

For the sake of simplicity, we use unary datamorphisms to explain the notion of combinatorial strategy. This can easily be extended to binary and n-ary datamorphisms for $n > 1$.

Assume that there is a set Ψ of $n > 0$ unary datamorphisms. A mutant test case m obtained by a sequence of applications of datamorphisms $\varphi_1, \varphi_2, \cdots, \varphi_l \in \Psi$ on a seed test case $s \in S$ is represented as

$$m = \varphi_1 \circ \varphi_2 \circ \cdots \circ \varphi_l(s).$$

A set T of test cases is said to be 1-way combinatorial complete, if for every seed test case s and every datamorphism φ, there is a test case $t \in T$ such that $t = \cdots \circ \varphi \circ \cdots (s)$. A set T of test cases is said to be 2-way combinatorial complete, if for any ordered pair of datamorphisms $\varphi_1, \varphi_2 \in \Psi$, for every seed test case s, there is a test case $t \in T$ such that $t = \cdots \circ \varphi_1 \circ \cdots \circ \varphi_2 \circ \cdots (s)$.

Similarly, we can define k-way combinatorial completeness for every $k > 2$. Moreover, we say a set of test cases is 0-way combinatorial complete, if it contains all seed test cases.

A set of test cases satisfies the k-way combinatorial coverage criterion, if it is n-way combinatorial complete for all $n = 0, \cdots, k$. Similarly, we can define k-way combinatorial completeness and k-way combinatorial coverage criteria for a set of non-unary datamorphisms.

As in traditional combinatorial testing, the number of test cases that satisfies the k-way combinatorial coverage criterion can be significantly smaller than the number of all combinations of k datamorphisms on all seed test cases. For many ML applications, the datamorphisms, like those for transformation of facial images, are often commutative, associative, and idempotent. Thus, the number of test cases to satisfy a combinatorial coverage criterion can be much smaller.

- Optimal

This strategy is inspired by genetic algorithms. Consider the set of seed test cases as the initial population and the datamorphisms as mutation operators. At each step in the test case generation process, select a subset of the current population and a subset of datamorphisms to generate new mutants and add them into the population. The selection can be guided by a fitness function to either achieve maximal fitness in a fixed number of cycles, or until the fitness level peaks, or until the population reaches a certain number. Depending on the definition of the fitness function and the choice of termination condition, various kinds of optimisation of the test set can be obtained.

- Random

A basic strategy is to select a seed test case and a datamorphism at random to generate one mutant a time until a total number of test cases is generated or the testing is adequate according to some adequacy criterion.

- Exploratory

For classification and clustering problems, a practical goal of testing is often to find out the boundary between two classes. This can be done by defining binary datamorphisms to seek the boundary points and thereby find the Pareto front. For example, assume that $P(x)$ is a program that classifies an input real number x into two classes A and B. If there are two test cases a and b such that $P(a) \neq P(b)$, a datamorphism $Mid(x, y) = (x + y)/2$ can be applied to generate a test case $c = Mid(a, b)$. If $P(a) \neq P(c)$, then another test case $d = Mid(a, c)$ will be generated and the program tested on that; otherwise, test case $d = Mid(b, c)$ is generated and tested. This process repeats iteratively until the distance between two test cases is small enough. In general, a test strategy may use the program output on test cases to determine which datamorphism to apply and on which test case. The principles of search-based testing apply [10].

IV. Experiment

In this section, we report an experiment to demonstrate the validity of datamorphic testing method.

A. Goal of The Experiment

The goal of the experiment is to investigate the validity of test case generation by applying datamorphisms on images for testing face recognition. The research questions to be answered are:

- *RQ1*: Is an image generated by applying datamorphisms on an existing image a valid test case for face recognition applications?
- *RQ2*: Are the test results on a ML application valid when using mutant test data obtained by applying datamorphisms in seed test cases?

B. Design of The Experiment

The experiment consists of three key elements: (a) the AI application to be tested; (b) the dataset used to select both the seed test cases and also the real test data used in comparison with mutant test cases; (c) the transformations on test cases to be used as the datamorphisms.

1) Applications under test: We have selected four real ML applications of the same kind from industry, i.e. four face recognition applications. They are:

1) Tencent Face Recognition[4]
2) Baidu Face Recognition[5]
3) Face++ online face recognition[6]
4) SeetaFace face recognition.

The first three are online services invoked through APIs written in Java. SeetaFace is an open source project based on openCV. The project is cloned from GitHub[7] and installed on our local computer system. It is written in C++ and our invocation code is also in C++.

These applications are used to evaluate the validity of test cases generated by datamorphisms on facial images as well as to demonstrate the applicability and cost effectiveness of the testing method. The employment of multiple ML applications of the same kind but developed by independent vendors enables us to demonstrate the reusability and generalisability of the datamorphisms and test strategies.

2) Datasets: Two public datasets are used in our experiment.

- *CelebA*[8], which contains ten thousand identities, each of which has twenty images. There are two hundred thousand images in total. We selected randomly 200 (i.e. 1%) images of different identities from the dataset as the seed test cases to generate mutant test cases.
- *PubFig*[9], which contains 58,797 images of about 200 people also collected from the internet. For each individual, the dataset contains multiple images (about 300 on average) taken in completely uncontrolled situations as non-cooperative subjects with large variation in pose, lighting, expression, scene, camera, imaging conditions and parameters, etc. This unique feature of the dataset makes it ideal to compare the images generated by applying datamorphisms.

3) Datamorphisms and Metamorphisms: Instead of manually operating the YouCam APP as in Section II, we used the open source GitHub project *AttGAN*[10], which implements a set of 13 facial attribute editing operators [7]. Each operator takes

[4]URL: https://ai.qq.com/product/face.shtml#detect
[5]URL: https://aip.baidubce.com/rest/2.0/face/v3/match
[6]URL: https://api-cn.faceplusplus.com/facepp/v3/compare
[7]URL: https://github.com/seetaface/SeetaFaceEngine
[8]URL: https://github.com/LynnHo/AttGAN-Tensorflow
[9]URL: http://www.cs.columbia.edu/CAVE/databases/pubfig/
[10]URL: https://github.com/LynnHo/AttGAN-Tensorflow

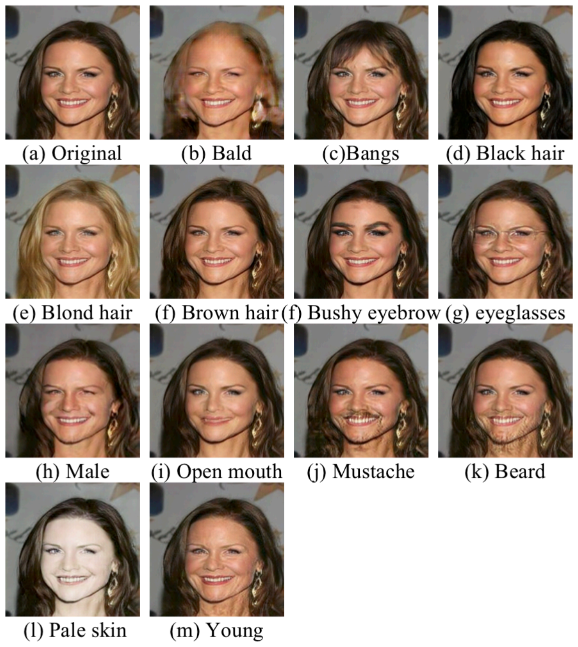


Fig. 3. Illustration of the image transformations.

a facial image as input and generates a new image that changes a facial attribute. They are listed in Table I and illustrated in Fig. 3.

TABLE I
ATTGAN'S FACE ATTRIBUTE EDITING OPERATORS

Operation	***Meaning***
Bald	Change the facial image into bald
Bangs	Add bangs to the facial image
Black Hair	Change the hair colour into black
Blond Hair	Change the hair colour into blond
Brown Hair	Change the hair colour into brown
Bushy Eyebrows	Change the eyebrows to be bushy
Eyeglasses	Add eyeglasses to the image
Male	Change the image from female to male
Mouth Open	Change the mouth to be slightly open
Mustache	Add or remove mustache to the facial image
Beard	Add or remove beard
Pale Skin	Make the skin tone to be pale
Young	Change the image to look younger

The metamorphisms used in the experiments are

$$FaceSimile(x, \varphi(x)) \geq 80\%$$

where $\varphi(x)$ is any of the datamorphisms given in Table I, *FaceSimile* is any of the four face recognition applications. For each of them, $FaceSimile(x, y)$ returns a number in the interval [0,100] as the similarity score between two facial images x and y.

C. Execution of The Experiment

The experiment consists of the following 3 steps.

1) Generation of mutant test cases: The mutant test cases are generated by using the AttGAN software on 200 images selected at random from the CelebA dataset.

The validity of the AttGAN algorithm for inverting facial images on various attributes has been intensively studied via cross-validation [7] on two large-scale labeled datasets, which clearly demonstrated that the resulting facial images achieved their purposes from the machine learning and image processing points of view. Our purpose differs from their experimental study. It is to validate the use of modified images as test cases for face recognition applications. Therefore, each facial image in the selection from the CelebA dataset is used as the seed, and 13 mutants generated by applying the transformations listed in Table I are used as mutant test cases. A total of 2,600 mutant test cases were generated.

2) Testing on generated test cases: The face recognition applications are tested on the mutant test cases against the seed test cases.

These mutants were input to four face recognition ML applications to obtain a measure of the similarity between the seed and the mutant, which is a numerical score in the range between 0 to 100. The raw data can be found in [8].

3) Testing on real images: To validate the result of the testing on these mutant test cases, we selected at random 13 images for each individual from the PubFig dataset. We use these real images to test the face recognition applications and obtained their recognition accuracies. A total of 2600 real images were used as the real test cases. See [8] for the test result data.

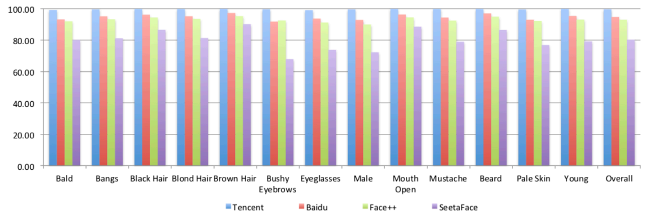
D. Analysis of The Results

The data collected from the experiments are analysed to answer the research questions RQ1 and RQ2.

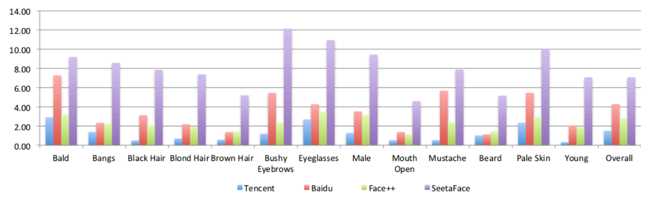
1) Validity of Using Generated Images as Test Data: To answer research question RQ1, we analyse how close the generated test cases are with respect to the original image. For each type of mutant, the average of similarity scores indicates how well the generated test case is close to the original image in the eyes of the ML application. The distribution of average similarity scores and their standard deviations over different image operators are shown in Fig. 4(a) and (b). Details of the data can be found in [8].

The results show that the overall average similarity scores are between 80.32 and 99.70 for different face recognition applications. The smallest standard deviation is 1.51 while the largest standard deviation is 7.07. Therefore, we can conclude that facial images generated by applying such image processing algorithms to change various attributes of the image are very close to the real images, thus valid as test cases.

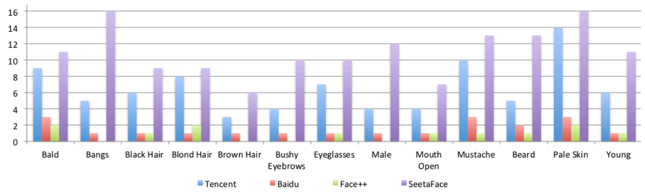
There are a small number of cases where recognition fails, as shown in Fig. 4(c). This is either because the application does not recognize any face in the image or because there is a timeout in the transmission of image data to the servers on the Cloud. Both of these cases are ignored when calculating recognition accuracy.



(a) Average Similarity Scores between Seeds and Mutants

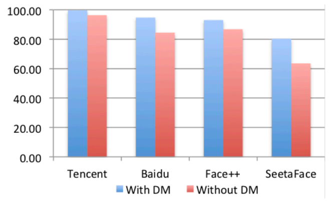


(b) Standard Deviations of Similarity Scores

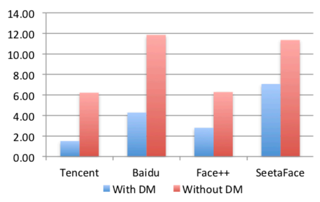


(c) Number of Failures

Fig. 4. Similarity between seeds and the mutants.



(a) Overall Average Scores



(b) Overall Std Deviations

Fig. 5. Testing on mutants vs on real test cases.

2) Test Effectiveness: To answer research question RQ2, we compare the test results obtained by using generated mutant test cases and the results obtained by using real images.

Our experiments demonstrated that using mutant test cases to test facial recognition applications can differentiate their recognition capability; see Fig. 5.

The results show that the Tencent Face Recognition gives the highest overall average (99.70) of similarity scores between the seeds and mutants, while SeetaFace has the lowest overall average (80.29) of similarity scores. Face++ and Baidu Face Recognition are very close on overall average scores, 93.09 and 94.75, respectively. These results are highly correlated to the overall average scores of the testing with real images. The Pearson's correlation coefficient between them is 0.99. The standard deviations are also highly correlated with a Pearson's correlation coefficient of 0.82. Therefore, we can conclude that the test results obtained by using mutant test cases is valid.

V. Conclusion

In this paper, we proposed a new software testing method called datamorphic testing and explored its applicability to testing ML applications. An experiment has shown it is a valid approach.

A. Related Work

The proposed approach is an improvement, generalisation and integration of many data centric testing methods.

Data mutation testing was proposed by Shan and Zhu to test software whose input is structurally complex [4]. A test framework in data mutation testing consists of a set of known test cases called seeds and a set of data mutation operators applicable on the seed test cases. This inspired the seed test cases and datamorphisms of our approach. An empirical study of data mutation testing method was conducted by testing a modelling tool. It demonstrated high fault detection ability. Datamorphic testing improves upon data mutation testing by integrating it with metamorphic testing. It advocates that data mutation operators should be engineered together with metamorphic relations, so that the correctness of the software on test cases can be automatically checked.

Metamorphic testing was proposed by Chen, et al. [5], who introduced the notion of metamorphic relations and explored its use in software testing. It has been an active research topic since then, with researchers empirically studying the effectiveness of the testing method. Recent surveys of the work on this topic can also be found [6].

Zhu has integrated data mutation testing and metamorphic test methods and showed that using data mutation operators, metamorphic relations can be easily derived [3], overcoming a long-standing barrier to practical use. Zhu also reported on an automated testing tool called JFuzz to support the use of integrated data mutation and metamorphic testing methods for Java unit testing. Datamorphic testing improves the applicability of metamorphic testing by allowing systematic derivation of datamorphisms and metamorphisms, as illustrated by the example above and the experiment. Moreover, JFuzz implements a simple fixed strategy of using data mutation operators and metamorphisms in testing. In contrast, the datamorphic testing method proposed in this paper regards the testing strategy as a variable element.

Fuzz testing is a type of random testing [9], in which a test case is randomly modified and the correctness criterion is that the software does not crash. It is a special trivial case of datamorphic testing. Thus, datamorphic testing can significantly increase testing effectiveness by specifying more meaningful modifications of the test cases with datamorphisms and more accurate correctness conditions in metamorphisms.

Search-based testing regards testing as an optimisation problem, to maximise the test effectiveness or test coverage by searching on the space of test cases [10]. Genetic algorithms are often employed to realise such optimisation. Datamorphic testing can be combined with search-based software testing to search for optimised test sets, for example, by using datamorphisms as means of generating a population of test cases, and coverage of metamorphisms as the optimisation target. As discussed in Section III, the principle of search-based testing can be applied to form specific testing strategies of datamorphism testing.

Testing ML applications is rarely reported in the research literature. An exception is DeepTest [2], which is a software tool for testing deep neural network (DNN) driven autonomous cars. It automatically generates test cases leveraging real world changes in driving conditions like rain, fog, lighting conditions, etc. via image transformations. Metamorphic relations are defined based on such image transformations and used to detect erroneous behaviours. DeepTest can be understood as a datamorphic testing tool for a special type of ML applications. It demonstrates that such synthetic test cases can be realistic and capable of finding a large number of erroneous behaviours under different realistic driving conditions, many of which led to potentially fatal crashes in three top performing DNNs in the Udacity self-driving car challenge. Most existing testing techniques for DNN-driven vehicles are heavily dependent on the manual collection of test data under different driving conditions. This is prohibitively expensive as the number of test conditions increases. DeepTest shows that the approach to generate a large number of realistic test cases can be cost efficient. However, it is unclear how the seed test cases reported in [2] are selected, how transformations on test cases are identified, and what test strategy were implemented in DeepTest to generate mutant test cases. These are some of the issues that this paper attempts to address.

The main contribution of this paper is a theoretical framework of a testing method for AI applications, which unifies the existing data centric testing methods and techniques. It lays a foundation for an engineering methodology and automated testing tool for AI applications.

B. Further Work

We are developing an automated testing tool to support datamorphic testing method for AI applications. It aims to show the feasibility of datamorphic testing with significant improvement in test effectiveness and efficiency. We are further investigating practical techniques in the framework of datamorphic testing. These techniques include (a) adequacy criteria for testing AI applications, (b) various test strategies and heuristics rules for controlling the testing activities, and (c) test process models that integrate testing with the development process of AI applications.

We are conducting further experiments with different testing strategies and empirical studies with different real AI applications in order to develop practical guidelines on the uses of the testing techniques and automated tools based on empirical evidence.

Acknowledgment

The work reported in this paper is partially supported by the National Science Foundation of China (Grant No. 61502233).

References

[1] X. Bai, J. Li, and A. Ulrich, (eds.), Proceedings of The 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST 2018), Gothenburg, Sweden, May 28, 2018.

[2] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars", in Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE 2018), Gothenburg, Sweden, 2018, pp. 303-314.

[3] H. Zhu, "JFuzz: A Tool for Automated Java Unit Testing based on Data Mutation and Metamorphic Testing Methods", in Proc. of the 2nd Int'l Conf. on Trustworthy Systems and Their Applications (TSA 2015), 8-9 July 2015, pp8-15.

[4] 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, vol. 52, no.5, pp571-588, Aug. 2009.

[5] T. Y. Chen, et al., "Metamorphic Testing: A New Approach for Generating Next Test Cases", Technical Report HKsUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[6] T. Y. Chen, et al., "Metamorphic Testing: A Review of Challenges and Opportunities", ACM Computing Surveys, vol. 51, no. 1, Article 4, 27 pages, January 2018.

[7] Z. He, W. Zuo, M. Kan, S. Shan, and X. Chen, "Arbitrary Facial Attribute Editing: Only Change What You Want", arXiv preprint arXiv:1711.10678, 2017.

[8] H. Zhu, D. Liu, I. 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, 21, 2018. *Available online at http://cms.brookes.ac.uk/staff/HongZhu/Publications/TR201802.pdf)*

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

[10] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications", ACM Computing Surveys, vol. 45. no.1, Article 11, 61 pages, November 2012.