
Article:
Patel, K. and Hierons, R.M. (2019) A partial oracle for uniformity statistics. Software Quality Journal, 27 (4). pp. 1419-1447. ISSN: 0963-9314

https://doi.org/10.1007/s11219-019-09459-0

Software Quality Journal
https://doi.org/10.1007/s11219-019-09459-0

A partial oracle for uniformity statistics

Krishna Patel[1] · **Robert M. Hierons**[1]

Published online: 20 August 2019

Abstract
This paper investigates the problem of testing implementations of uniformity statistics. In this paper, we used metamorphic testing to address the oracle problem of checking the output of one or more test executions, for uniformity statistics. We defined a partial oracle that uses regression analysis (a regression model–based metamorphic relation). We investigated the effectiveness of our partial oracle. We found that the technique can achieve mutation scores ranging from 77.78 to 100% and tends towards higher mutation scores in this range. These results are promising and suggest that the regression model–based metamorphic relation approach is a viable method of alleviating the oracle problem in implementations of uniformity statistics, and potentially other classes of statistics, e.g. correlation statistics.

Keywords Uniformity statistics · Oracle problem · Non-testable systems · Metamorphic testing · Regression model–based metamorphic relation · Metamorphic relation

1 Introduction

Testing is a verification process that consists of the following steps: (1) generate a test case, (2) predict the outcome of the test case, (3) execute the system under test (SUT) with the test case to obtain the actual outcome and (4) compare the predicted outcome against the actual outcome to obtain a pass/fail verdict. The *oracle problem* describes scenarios in which either it is impossible to carry out steps 2 or 4 or these are prohibitively expensive. This issue was initially discussed by Weyuker, who used the term 'untestable programs' (Weyuker 1982). It has been observed that many real systems suffer from the oracle problem, such as bioinformatics systems (Chen et al. 2009), service-oriented architectures (Chan et al. 2007) and systems that implement graph theory, computer graphics and compilers (Zhou et al. 2004). One might expect a system to suffer from the oracle problem if the required relationship between the input and output is highly complex and the problem being solved by the system is such that we do not know the answer in advance.

✉ Krishna Patel
krishna.patel@sheffield.ac.uk

Robert M. Hierons
r.hierons@sheffield.ac.uk

[1] Department of Computer Science, The University of Sheffield, 211 Portobello, Sheffield, S1 4DP, UK

Since the oracle problem is a pervasive and non-trivial problem, it has inspired a large amount of research. Numerous techniques have been proposed as solutions to the oracle problem, including assertions, metamorphic testing, statistical hypothesis testing, N-version testing and machine learning oracles (Patel and Hierons 2017). However, the main focus has been on *metamorphic testing*, an approach/term introduced by T.Y. Chen (1998, 2003).

Metamorphic testing (MT) is based on a conceptually simple but powerful idea. Essentially, in metamorphic testing we have a *metamorphic relation* that is an expected property of the SUT, with this property mentioning a number of inputs and corresponding outputs. Let us suppose, for example, that we are to test an implementation f of the sine function. We have a number of properties of this, such as $\sin(x) = -\sin(-x)$. Thus, for example, if $f(0.346) \neq -f(-0.346)$ then we can deduce that f is faulty even though we do not know the values of $\sin(0.346)$ and $\sin(-0.346)$. In metamorphic testing, we typically run the SUT on a number of inputs $x_1, \ldots, x_k$, observe outputs $y_1, \ldots, y_k$, and on the basis of these, we then choose a next input[1] x_{k+1} to use. We then apply x_{k+1}, record the resultant output y_{k+1} and check that the inputs $x_1, \ldots, x_{k+1}$ and outputs $y_1, \ldots, y_{k+1}$ satisfy the metamorphic relation (for recent surveys on MT, see Chen et al. (2018) and Segura et al. (2016)).

In this paper, we consider the problem of testing an implementation of a uniformity statistic. A sample is a collection of data points, such as $\text{Sample}_1 = \{1, 2, 2, 3, 4, 5\}$ or $\text{Sample}_2 = \{1, 2, 3, 4, 5, 6\}$, that was produced by some stochastic process such as a random number generator. The distribution of the stochastic process is said to be uniform if every element has an equal chance of being randomly selected. Uniformity statistics are a means of measuring the extent to which a sample is uniformly distributed. Our initial interest in uniformity statistics was motivated by two aspects of software engineering. First, it has been observed that test suites with highly diverse test cases are often effective in finding faults, at least when compared with less diverse test suites (see, for example, Feldt et al. 2008). There are test generation approaches, such as adaptive random testing (Chen et al. 2004), that aim to generate highly diverse test suites but if we can measure uniformity then there is potential, for example, to incorporate a uniformity statistic as a fitness function in search-based software testing. The second motivation is that in experiments we would like to uniformly sample from a space of subjects/examples; we might use uniformity statistics to either check the sampling mechanism used or to drive sampling. Note also that if we can test for uniformity then we can also test samples against other types of distributions (Marhuenda et al. 2005).

Uniformity statistics play an important role in many software engineering research areas. For example, in the context of software quality, Chen (2013) used the Kolmogorov-Smirnov test to evaluate the uniformity of measures of defect density. In the context of software testing, Dutra et al. (2018) developed an algorithm that can generate a set of test cases, such that this set is uniformly distributed, and each test case in the set satisfies some constraint. They used the χ^2 uniformity statistic to evaluate their tool.

Consider a genetic algorithm that implements a crossover operator with a crossover probability of 40%. This crossover operator might be implemented as follows: a number between 0 and 1 is randomly sampled from a uniform distribution; if this number is less than or equal to 0.4, then the crossover operator will be invoked. A fault might exist in this implementation that causes the distribution to be non-uniform; in particular, the distribution might be positively skewed. The existence of such a fault would mean that the probability of crossover

[1] There may be more than one 'next input'.

is greater than 40%, and assuming that 40% is the optimal crossover rate, this would lead to the genetic algorithm performing suboptimally. This demonstrates that the correctness of uniform distributions is important for tools and techniques that rely on them and by implication software engineering research that uses these tools. Examples of such tools include random number generators and Monte Carlo simulations (Harrison 2010), adaptive random testing (Liu et al. 2010), an algorithm that was proposed by Claessen et al. (2014) that generates test data that complies with the uniform distribution and a set of constraints, the cross entropy–based test case generation approach proposed by Chockler et al. (2007) and metrics for measuring information loss in a system, which were developed by Androutsopoulos et al. (2014). Confidence in the correctness of a uniform distribution can be ascertained by uniformity statistics.

Implementations of uniformity statistics suffer from the oracle problem because it is often infeasible to predict the expected value of a test statistic. This paper investigates a metamorphic testing and machine learning–based approach for overcoming the oracle problem for this class of SUT. Metamorphic testing was chosen because it has been found to be effective in many areas (Chen et al. 2018; Segura et al. 2016). We believe that implementations of many other types of statistics (e.g. correlation statistics) also suffer from the oracle problem, and we hope that the work outlined in this paper will feed into the development of testing techniques for other such statistics. We are not aware of any work that has used MT for the testing of implementations of statistics. However, MT has been applied to the problem of checking the output of stochastic programs (Guderlei and Mayer 2007). In this previous work, statistical hypothesis testing was used to check properties (e.g. mean, standard deviation) of samples produced with different (but related) inputs.

Others have investigated means of enhancing metamorphic testing with the use of machine learning techniques. For example, Chan et al. (2010) formulated a hybrid technique, where a machine learning oracle is applied to a set of test cases. If this oracle detects a failure, then the hybrid approach concludes that the system is faulty. However, if this oracle does not detect a failure, then the test cases that were evaluated by this oracle are then used for metamorphic testing. Another example includes Kanewala and Bieman's technique (Kanewala and Bieman 2013, 2015), which uses machine learning to identify whether target code exhibits a particular metamorphic relation. Such applications of machine learning in the context of metamorphic testing differ from our own.

The approach developed in this paper is motivated by the following observation: a normal distribution, with a mean of 0.5, which has a higher standard deviation, has a 'flatter' distribution in the region [0, 1], and so samples that are drawn from such distributions in the range [0, 1] should adhere more strongly to uniformity. In other words, there is a positive correlation between standard deviation and uniformity, when considering normal distributions. Regression analysis (which is a form of machine learning) can be used to build predictive models based on such correlations. We developed a metamorphic relation, which we call the regression model–based metamorphic relation, which operates by comparing the implementation of a uniformity statistic to such a predictive model; comparisons that reveal a discrepancy between the implementation and model are an indication that a fault might be present in the uniformity statistic.

This approach can be seen as an instance of metamorphic testing, if one uses the recent generalised definition of metamorphic testing (Chen et al. 2018), since we utilise the results of previous test executions in checking the outcome of a test, assuming that the training data is obtained from the SUT. Training data might be obtained from the SUT in the context of regression testing, or might be obtained by other means e.g. from a reference implementation, in other contexts.

The proposed technique was evaluated by applying it to implementations of 18 different uniformity statistics. As we explain in Section 4, these uniformity statistics were chosen since they are representative of different approaches to checking for uniformity. The results of the evaluation were promising; the proposed technique obtained an average mutation score of 92.96% and can obtain a false positive rate of 0, and the predictive models were reasonably accurate.

The main contributions made in this paper are:

1. A new type of metamorphic relation called regression model–based metamorphic relation
2. 18 regression model–based metamorphic relations for uniformity statistics
3. An investigation of the effectiveness of regression model–based metamorphic relations for uniformity statistics.

The remainder of this paper is structured as follows. Section 2 outlines background material, and Sections 3 introduces the proposed technique. Section 4 describes the experimental design; Section 5 presents the results of the experiments, and Section 6 explores threats to validity. Finally, Section 7 draws conclusions and outlines future work.

2 Background

2.1 The oracle problem

The oracle problem refers to situations in which it is infeasible to predict the test outcome or verify the program's output against the predicted test outcome (Weyuker 1982). A number of techniques have been proposed to alleviate the oracle problem, including metamorphic testing, statistical hypothesis testing, N-version testing and machine learning oracles (Patel and Hierons 2017; Chen et al. 1998, 2003). This section briefly introduces these techniques.

A *metamorphic test group* (MG) is a sequence of test cases (Chen et al. 2018). Each test case in a metamorphic test group can be classified as either a *source test case* or a *follow-up test case*. Source test cases are produced by some, possibly arbitrary, test case generation method, whilst follow-up test cases are generated based on source test cases and possibly also the response of the SUT to these inputs (Chen et al. 2018). For example, consider a sorting algorithm. A source test case, tc_s, might be a randomly generated sample, and a follow-up test case, tc_f, might be the result of reversing the order of tc_s. A *metamorphic relation* (MR) is an expected relation between source and follow-up test cases and their resulting outputs. For example, one would expect that the sorting algorithm would produce the same outputs for tc_s and tc_f.

There are a number of factors that can affect the effectiveness of a metamorphic relation (Patel and Hierons 2017). For example, Cao et al. (2013) discovered that the diversity of paths taken by test cases in a metamorphic test group is an important determinant of effectiveness. Another factor is the *tightness* of an MR (Liu et al. 2014), which refers to how much an MR constrains behaviour. For example, suppose that two MRs, MR_t and MR_l, were developed for the max function, $\max(a, b)$. MR_t might check that $\max(a, b) \times 2 = max(a \times 2, b \times 2)$, whilst MR_l might check $\max(a, b) < \max(a \times 2, b \times 2)$ (for positive a and b). MR_t is tighter than MR_l because if we fix one side of the equation then only one value can satisfy MR_t, whilst an infinite number of values could satisfy MR_l. Finally, if we use multiple MRs then it is desirable to use a diverse set of MRs (Liu et al. 2014; Chen et al. 2015).

N-version testing, which is also referred to as back-to-back testing, involves comparing the outputs of the SUT against the outputs of reference implementations (Weyuker 1982). One of the main issues is that a reference implementation may not be available. A number of approaches have been developed to overcome this obstacle, e.g. testability transformations (McMinn 2009), component harvesting (Hummel et al. 2006) and using previous versions of the SUT as reference implementations (Zhang et al. 2009).

The third approach that we consider in this section is statistical hypothesis testing. Statistical hypothesis testing involves executing the SUT multiple times to obtain a set of outputs, and then conducting a statistical test, e.g. t test, to compare the distribution of this set to an expected distribution (Guderlei and Mayer 2007). One issue with this approach is that it assumes that the expected distribution is already known (Mayer 2005), which might not necessarily be true. It is sometimes possible to combine this with MT, by comparing the properties of samples produced using different (but related) inputs (Guderlei and Mayer 2007). Another issue is the need to tune parameters, e.g. it is necessary to tune the significance threshold of statistics that are used by a statistical hypothesis test (Patel and Hierons 2017).

Finally, machine learning oracles are predictive models or classifiers that can be used as test oracles (Patel and Hierons 2017). For example, a machine learning oracle might be an approximation of the SUT and so can predict the output of a given input or might be able to classify a test case as passed or failed, based on patterns that can be observed in the execution trace. Machine learning oracles are derived from machine learning algorithms that have been applied to training datasets. Similar issues to those that were discussed for statistical hypothesis testing are relevant for machine learning oracles (Patel and Hierons 2017). For example, some machine learning oracles have tunable parameters (Patel and Hierons 2017). One also requires a training dataset; again one might alleviate this assumption by procuring a dataset from a domain expert or a reference implementation (Chan et al. 2006).

2.2 Uniformity statistics

In this paper, we considered five statistics that are *empirical distribution function statistics*: D_n^+, D_n^-, V_n, W_n^2 and U_n^2. For a given value x, an empirical distribution function[2], which is associated with a particular sample, returns the probability that a value that is chosen at random from the associated sample will be less than or equal to x. Empirical distribution function statistics operate by comparing the cumulative distribution function to the empirical distribution function (Marhuenda et al. 2005).

We also considered eight *order statistics* (Marhuenda et al. 2005): C_n^+, C_n^-, C_n, K_n, T_1, T_2, $T_1^{'}$ and $T_2^{'}$. Let Sample$_o$ be a sample $\{U_{(1)}, U_{(2)}, ..., U_{(n)}\}$, which has been arranged in ascending order. The core intuition behind order statistics is to compare each element of the sample $U_{(i)}$ with its expected value. In the case of the former 6 statistics, the expected value of $U_{(i)}$ is defined as $i/(n+1)$, where n is the sample size. The expected value of $U_{(i)}$ for the latter two statistics is $(i-1)/(n-1)$.

Spacings statistics ($G(n)$ and Q) are computed based on spacings (Marhuenda et al. 2005). Spacing is defined to be the difference between two adjacent elements in an ordered (ascending order) sample. There are two exceptions to this; firstly, a spacing is defined to be the value of the first element of the sample, if the two adjacent elements are the first

[2]Empirical distribution functions are also referred to as 'empirical cumulative distribution functions".

element of the sample and the non-existent element that is adjacent left of the first element, e.g. in the case of Sample$_o$, the spacing for $U_{(0)}$ and $U_{(1)}$ would be $U_{(1)}$. Similarly, if the two adjacent elements are the last element of the sample, and the non-existent element that is adjacent right of the last element, then the spacing is the difference between 1 and the value of the last element of the sample. In continuation of the previous example, the spacing for $U_{(n)}$ and $U_{(n+1)}$ would be the difference between 1 and $U_{(n)}$.

$S_n^{(m)}$, $A^*(n)$ and $E_{m,n}$ are uniformity statistics that are referred to as *higher order spacing statistics*. In the context of higher order spacings statistics, a spacing is defined to be the difference between two elements from an ordered (ascending order) sample. Unlike the definition of a spacing used by the spacings statistics, the elements are not necessarily adjacent—they are separated by $m - 1$ elements, where m is a user-defined parameter. To illustrate, consider the following sample and value for m: sample $= \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $m = 3$ respectively. Examples of spacings include the difference between elements 1 and 4, 2 and 5, and 3 and 6 (because in all cases, there is a gap of $m - 1$ i.e. 2 elements).[3] Higher order spacing statistics are computed based on spacings (Marhuenda et al. 2005).

3 Regression model–based metamorphic relation

3.1 Intuition

Let US denote an implementation of a uniformity statistic. Also let training data be a set of pairs, $\langle SD_i, TSV_i \rangle$, such that SD_i is the standard deviation of a sample, Samp$_i$, and TSV_i is the result of computing US on Samp$_i$. A predictive model, PM_{US} can be obtained through a regression analysis of training data. PM_{US} can predict the output of US for a given standard deviation value.

Our partial oracle operates by first generating a set of samples and executing US with these samples to obtain a set of outputs. It then computes the standard deviation of each of these samples and executes PM_{US} with these standard deviations to obtain another set of outputs. Finally, it uses the Wilcoxon signed rank test to compare the first set of outputs against the second set of outputs. A significant difference indicates failure.

Our partial oracle is an example of a metamorphic relation. The remainder of this section illustrates this by superimposing metamorphic testing jargon on to the description above. PM_{US} can be derived from information about previous executions; thus, inputs for PM_{US} can be conceptualised as source test cases. Inputs into US are akin to follow-up test cases. A comparison is performed between the outputs of the source and follow-up test cases (facilitated by the Wilcoxon signed rank test), to determine whether the system is faulty.

3.2 The approach

The standard deviation is intrinsically linked to the values of test statistics. We realised that it might be possible to use regression analysis,[4] which is a form of machine learning, to learn the precise nature of the relationship between the standard deviation and a given test statistic and derive a formula that describes the relationship.

[3] Throughout this paper, we set $m = 3$. This choice of m was deemed to be appropriate for our samples sizes.
[4] We used Microsoft Excel to carry out the regression analysis.

Table 1 Mathematical models

Statistic	Mathematical model
C_n^-	$2.2662 \times \text{StandardDeviation}^2 - 2.1899 \times \text{StandardDeviation} + 0.4863$
C_n^+	$2.214 \times \text{StandardDeviation}^2 - 2.1799 \times \text{StandardDeviation} + 0.4861$
C_n	$2.1619 \times \text{StandardDeviation}^2 - 2.1431 \times \text{StandardDeviation} + 0.4866$
W_n^2	$912.54 \times \text{StandardDeviation}^2 - 550.32 \times \text{StandardDeviation} + 82.977$
$S_n^{(m)}$	$569048 \times exp(-16.13 \times \text{StandardDeviation})$
D_n^-	$2.2127 \times \text{StandardDeviation}^2 - 2.1802 \times \text{StandardDeviation} + 0.4871$
D_n^+	$2.2648 \times \text{StandardDeviation}^2 - 2.1902 \times \text{StandardDeviation} + 0.4873$
$G(n)$	$-52.351 \times \text{StandardDeviation}^3 + 33.843 \times \text{StandardDeviation}^2$ $-7.2088 \times \text{StandardDeviation} + 0.5087$
$T_1^{'}$	$-0.2665 \times \text{StandardDeviation}^2 - 0.7551 \times \text{StandardDeviation} + 0.2491$
$T_2^{'}$	$0.9125 \times \text{StandardDeviation}^2 - 0.5509 \times \text{StandardDeviation} + 0.0831$
K_n	$4.4802 \times \text{StandardDeviation}^2 - 4.3698 \times \text{StandardDeviation} + 0.9724$
$E_{m,n}$	$0.3241 \times \text{StandardDeviation}^{-1.014}$
Q	$-44.946 \times \text{StandardDeviation}^3 + 30.612 \times \text{StandardDeviation}^2$ $-6.8491 \times \text{StandardDeviation} + 0.508$
$A^*(n)$	$-22015 \times \text{StandardDeviation}^3 + 15305 \times \text{StandardDeviation}^2$ $-3454.2 \times \text{StandardDeviation} + 254.07\ T_1\ -0.8204 \times \text{StandardDeviation} + 0.2513$
T_2	$0.9126 \times \text{StandardDeviation}^2 - 0.5498 \times \text{StandardDeviation} + 0.0828$
V_n	$4.4775 \times \text{StandardDeviation}^2 - 4.3704 \times \text{StandardDeviation} + 0.9744$
U_n^2	$912.05 \times \text{StandardDeviation}^2 - 550.42 \times \text{StandardDeviation} + 82.98$

A test case for a uniformity statistic is a sample that contains 1000 elements, drawn from a normal distribution[5] parameterised with a mean of 0.5 and a standard deviation between 0 and 0.28, such that each element is a value between 0 and 1. We generated 20,000 test cases and calculated the standard deviation of each test case (sample). Our rationale for selecting these values can be found in Section 4.4.

The following procedure can be used to acquire a formula that describes the relationship between the test statistic and the standard deviation, for a given uniformity statistic. We evenly divided the range 0 to 0.3 into 60 subranges Subranges = {[0, 0.005], [0.005, 0.01], ..., [0.295, 0.3]} (these subranges were chosen because they enable effective regression analysis to be conducted for our dataset). Each test case was assigned to a subrange, such that the standard deviation of the test case was greater than or equal to the lower bound of the subrange and less than the upper bound of the subrange. We calculated the test statistic values (using our implementations of the statistics) of all of the test cases in a given subrange and averaged these values. This average test statistic value was then associated with the midpoint of the subrange, e.g. 0.0025 is the midpoint of the first subrange in Subranges. This was repeated for all of the subranges. Finally, a regression analysis was performed based on these averages and midpoints, resulting in a mathematical formula that gives a relationship between the test statistic and the standard deviation.

[5]Implemented in Python using the numpy random.normal method.

We adopted this procedure for each test statistic (see Table 1), with the exception of $E_{m,n}$, where the subrange with the range of 0–0.005 was omitted, since the behaviour in this range is uncharacteristic; excluding this subrange improved the accuracy of the regression analysis.

These formulas served as predictive models (partial oracles) that can be used to check the output of the implementation of a test statistic.

3.3 Using the regression model based metamorphic relation

Algorithm 1 Regression model based metamorphic relation.

Input: The Regression Model based Metamorphic Relation accepts values for the following variables as input: $SizeLowerbound$, $SizeUpperbound$, $Distribution$, $MeanLowerbound$, $MeanUpperbound$, $StandardDeviationLowerbound$, and $StandardDeviationUpperbound$. A test case generator, TCG, can be used to generate a random sample based on these variables. In particular, it can draw between $SizeLowerbound$ and $SizeUpperbound$ elements from a $Distribution$ that has been parameterised with a mean that is between $MeanLowerbound$ and $MeanUpperbound$, and a standard deviation that is between $StandardDeviationLowerbound$ and $StandardDeviationUpperbound$, such that each element is a value between 0 and 1. The Regression Model based Metamorphic Relation also accepts a value for the following variable as input: $NumberOfTestCases$

Output: A "Passed" or "Failed" verdict.

```
1  Let TestStatisticValues be an empty list;
2  Let ModelValues be another empty list;
3  while TestStatisticValues.size() != NumberOfTestCases do
4  |   Sample = TCG.generateSample();
5  |   TestStatisticValue = evaluateUniformityStatistic(Sample);
6  |   TestStatisticValues.add(TestStatisticValue);
7  |   ModelValue = evaluateModel(calculateStandardDeviation(Sample));
8  |   ModelValues.add(ModelValue);
9  end
10 isSignificant = WilcoxonSignedRank(TestStatisticValues, ModelValues);
11 if isSignificant == True then
12 |   return "Failed";
13 else
14 |   return "Passed";
15 end
```

Algorithm 1 describes the regression model–based metamorphic relation. Lines 1 and 2 of algorithm 1 create two empty lists, TestStatisticValues and ModelValues. Lines 3–9 then generate NumberOfTestCases test cases and, for each of these test cases, calculates the test statistic value and model value and stores them in the aforementioned lists. Thus, in the program state of the regression model based metamorphic relation, just before the execution of line 10, TestStatisticValues and ModelValues will contain NumberOfTestCases test

statistic values and NumberOfTestCases model values respectively. Line 10 then performs a Wilcoxon signed rank test to compare TestStatisticValues and ModelValues, to determine whether the difference is statistically significant. Finally, lines 11–15 report a failure verdict, if the difference is statistically significant or otherwise report a pass verdict.

4 Experimental design

The experiments described in this section were designed to address the research questions in Section 4.1.

4.1 Research questions

RQ1. How accurate are the models that are used by the regression model–based metamorphic relation? We inspected the accuracy of the models, to gain insights into the effectiveness of the regression model–based metamorphic relation.

RQ2. How many false alarms does our technique raise? We evaluated the soundness of the partial oracle, based on its false positive rate, i.e. the likelihood that it will incorrectly report that failure is present.

RQ3. How effective is the regression model–based metamorphic relation at detecting faults? We measured the effectiveness of our technique, by quantifying the proportion of mutants it kills.

RQ4. To what extent does the output of the mutant have to deviate from the output of the model, in order for our technique to detect the difference? We estimated the sensitivity of the regression model–based metamorphic relation.

RQ5. What impact would changing the alpha value have on the results? We conducted a sensitivity analysis to determine the impact of changing the alpha value.

RQ6. Why didn't the regression model–based metamorphic relation achieve a mutation score of 100%? The partial oracle did not kill all of the mutants. We attempted to identify the main root causes.

4.2 Experimental subjects

Our experimental subjects were implementations of 18 uniformity statistics (see Section 2) that were programmed in Python 3.6 by the authors, based on the technical details that were expressed in an article authored by Marhuenda et al. (2005). The Appendix gives pseudocode for these statistics. These were chosen because they are examples of four different types of statistics (Marhuenda et al. 2005).

The representativeness of the experiment was the key motivating factor for our choice of uniformity statistics. In particular, a sample of 18 statistics was deemed to be large enough to enable conclusions about generalisability to be drawn. Additionally, these statistics are representative of several different types of uniformity statistics, i.e. empirical distribution function, order, spacings and higher order spacing statistics (Marhuenda et al. 2005).

Table 2 shows the number of lines of code in each subject program. The sizes of these subject programs are representative of the typical size of an implementation of a uniformity statistic.

Table 2 Number of lines of code (LOC) in each subject program

Subject program	LOC	Subject program	LOC	Subject program	LOC
C_n	46	D_n^+	24	Q	40
C_n^-	25	$G(n)$	25	$A^*(n)$	28
C_n^+	25	T_1'	24	T_1	25
W_n^2	24	T_2'	24	T_2	25
$S_n^{(m)}$	28	K_n	54	V_n	41
D_n^-	24	$E_{m,n}$	29	U_n^2	30
(a)		(b)		(c)	

4.3 Faults

Mutation testing is a technique that can make small syntactic changes to a system (Offutt 1992) to simulate a fault. For example, suppose that mutation testing was applied to a system Sys_o and that the technique modified $abs(x) + 4$ to $abs(x) - 4$; let Sys_n denote the modified version of Sys_o. In mutation testing, $abs(x) - 4$ is referred to as a *mutation*, Sys_n is called a *mutant*, and if a testing technique reports a failure whilst being evaluated on Sys_n, Sys_n is said to have been *killed*, or is otherwise said to have *survived*. Mutation testing is a widely used method for procuring large samples of mutants, and some evidence suggests that mutants are representative of real faults (Andrews et al. 2005).

We applied a well-known, automated mutation testing tool called Mutmut (Hovmöller 2017) to all of the subject programs, to obtain a sample of mutants. In the context of black box testing, a mutant that has the same input-output mapping as the original program for a given test suite is equivalent to the original program with respect to that test suite. Such mutants could confound the results because it is not possible for an oracle to detect a difference between the original and mutant programs; thus, these were removed from the sample. As will be discussed in Section 4.4, we used a total of 5000 test cases. These 5000 test cases were used to conduct strong mutation testing on each mutant. Mutants were classified as equivalent if they were not killed by strong mutation testing. We also removed mutants that crash and time out (120 seconds), because testing techniques are not required to detect such mutants, and so these mutants can confound the results. After removing the equivalent, crashed and timed out mutants, a total of 291 mutants remained. We used all of these mutants in our experiments.

Tables 3 a, b and c show how the mutants are distributed across the subject programs.

4.4 Test suites

We initialised the regression model–based metamorphic relation with the following input values:SizeLowerbound = 1000, SizeUpperbound = 1000, Distribution = Normal, MeanLowerbound = 0.5, MeanUpperbound = 0.5, StandardDeviationLowerbound = 0, StandardDeviationUpperbound = 0.28 and NumberOfTestCases = 5000 (see algorithm 1).

Since the regression model–based metamorphic relation for uniformity statistics exploits the relationship between the standard deviation and test statistic value, it is natural to restrict oneself to samples that only vary in terms of the standard deviation. This was our rationale

Table 3 Number of mutants for each subject program

Statistic	Mutants	Statistic	Mutants	Statistic	Mutants
$A^*(n)$	25	$E_{m,n}$	17	T_1'	15
C_n	16	$G(n)$	16	T_2	15
C_n^-	9	K_n	17	T_2'	17
C_n^+	9	Q	27	U_n^2	27
D_n^-	9	$S_n^{(m)}$	16	V_n	14
D_n^+	7	T_1	13	W_n^2	22
(a)		(b)		(c)	

for initialising the regression model–based metamorphic relation, such that the sample size, distribution and mean are the same across all samples, but the standard deviation varies.

Our reasons for choosing the specific values for the sample size, distribution, and mean and standard deviation are as follows. A sample size of 1000 was chosen, because large samples are more likely to achieve the target standard deviation. Many statistical tests assume a normal distribution; thus, generating samples that conform to a normal distribution could enable us to experiment with alternative statistical tests. Since every element in a given sample is a value between 0 and 1, it was natural to fix the mean at 0.5—the midpoint of an element's value range. Recall that the regression model–based metamorphic relation samples values in the range [0, 1]; this restriction makes the random generation of samples with higher standard deviation values increasingly unlikely. Our choice of the standard deviation's upper bound was informed by this observation. As an aside, we should note that the uniformity statistics that are covered in this paper assume that the sample is being drawn from [0, 1], which is why our sample elements are always restricted to values in the range [0, 1].

We decided to use a test suite that consisted of 5000 test cases, because this test suite size was deemed to be large enough to enable us to draw meaningful conclusions. It should be noted that the same 5000 test cases were used for each subject program, and that these test cases are different from the ones that were used in Section 3.2.

4.5 Measures

One of the main measures that was used throughout this paper is the mutation score (MS) (see, for example, Wu et al. 2005). The mutation score can be calculated as follows (equivalent mutants were not counted):

$$\frac{\text{NumberOfKilledMutants}}{\text{TotalNumberOfMutants}}$$

We also used the false positive rate (FPR). The FPR can be determined by executing the test suite on a correct version of the SUT, recording the number of test cases that reported failures, NumberOfFalsePositiveTestCases and finally calculating:

$$\frac{\text{NumberOfFalsePositiveTestCases}}{\text{TestSuiteSize}}$$

4.6 Alpha

Recall that the regression model–based metamorphic relation performs a significance test (see algorithm 1). We used an alpha value of 0.000005 for this significance test. Our motivation for choosing such a low significance threshold is to reduce the sensitivity of the test to account for potential inaccuracies in the models.

5 Results and discussion

In this section, we address the research questions that were outlined in Section 4.1.

5.1 RQ1. How accurate are the models that are used by the regression model based metamorphic relation?

Each figure in Fig. 1a–r is a scatterplot for one subject program. Each interval of the X-axis pertains to a test case, and intervals of the Y-axis correspond to uniformity statistic values. Black dots on the scatterplot represent the correct test statistic value and grey dots represent the model's prediction.

Figure 1 a–r reveal that for many subject programs, the test statistic and model are almost indistinguishable, e.g. W_n^2 (Fig. 1d). These models are therefore incredibly accurate. Recall that the models were created by analysing the relationship between the test statistic value and one factor, i.e. the standard deviation. The existence of cases where the statistic and model are less similar, e.g. $S_n^{(m)}$ (Fig. 1e), suggests that other factors might also be important. Thus, there may be scope to improve the accuracy of the models by considering such factors. Interestingly however, despite the dissimilarities between the statistics and models in these cases, one can observe that the grey dots largely overlap with the cluster of black dots. This indicates that the model can provide reasonably accurate predictions for even these statistics.

We conducted one Mann-Whitney U test per scatterplot in Fig. 1a–r to compare the actual values against the predicted values. We also applied Benjamini-Hochberg correction to all of the p values that were returned by these tests. Promisingly, with the exception of Mann-Whitney U tests for four uniformity statistics ($S_n^{(m)}$, $G(n)$, Q and $A^*(n)$), none of the Mann-Whitney U tests reported a significant difference. This supports our observation that most models are relatively accurate, and that some models are more accurate than others.

The inaccuracy of some models may make partial oracles that are based on these models more susceptible to false positives. We investigate this possibility in the next section.

5.2 RQ2. How many false alarms does our technique raise?

Only four of the subject programs reported false positives—$T_2^{'}$, $E_{m,n}$, Q and $A^*(n)$. An investigation revealed that the cause of these false positives was an inappropriate use of alpha values. We found that changing the significance criteria to be less than 3.8921426976752E-39, 1.09300719274052E-12, 2.08955179387564E-07 and 4.13728711820944E-09, for $T_2^{'}$, $E_{m,n}$, Q and $A^*(n)$ respectively, eliminates all false positives and makes no difference to any mutant classifications for $T_2^{'}$, Q and $A^*(n)$ and only affects 2/17 of the mutant classifications for $E_{m,n}$ (these mutant classifications will be discussed in the next section). These results demonstrate that the appropriate choice of alpha value is important.

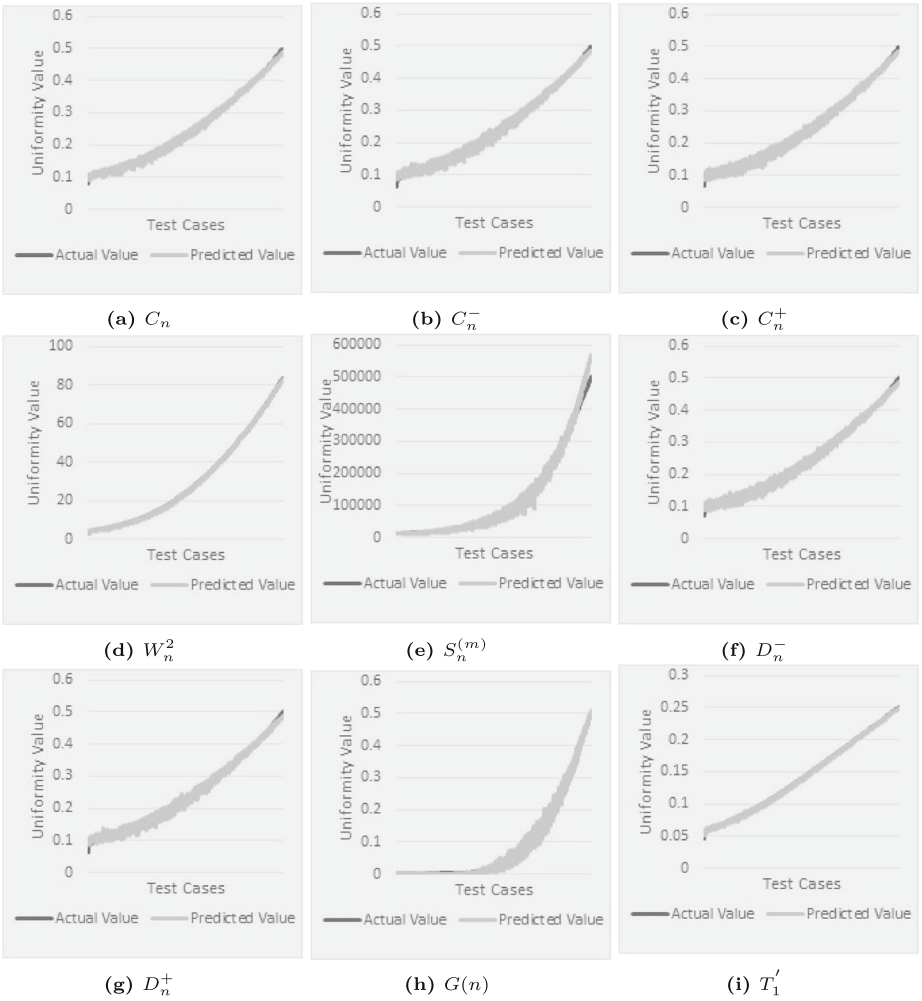
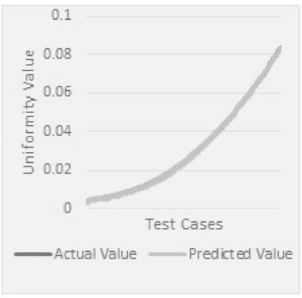


Fig. 1 A series of comparisons between each statistic and its respective model

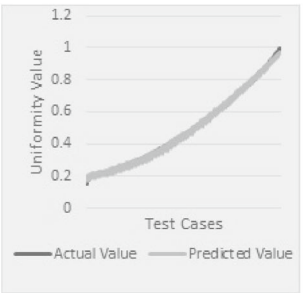
Given that our approach incorporates stochastic sample generation, it is possible for it to produce and process extreme/uncharacteristic samples. Such samples could potentially cause false positives. The above results demonstrate that, given an appropriate choice of alpha value, the technique is very robust despite this. This suggests that the probability of such samples being generated is low and/or that such samples are unlikely to have significant influence. This can be attributed to the large number of samples that were used and the large sizes of these samples.

5.3 RQ3. How effective is the regression model based metamorphic relation at detecting faults?

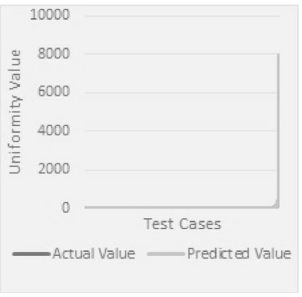
Figure 2 is a bar chart in which each interval of the Y-axis pertains to one subject program. The X-axis communicates the number of mutants that were used to exercise the regression



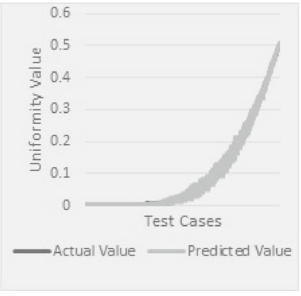
(j) T_2'



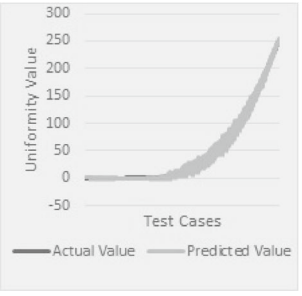
(k) K_n



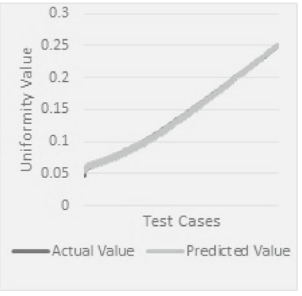
(l) $E_{m,n}$



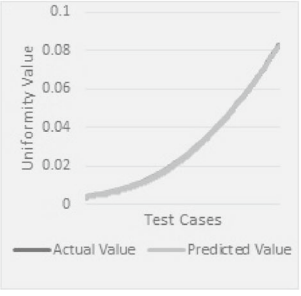
(m) Q



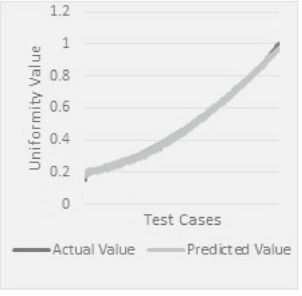
(n) $A^*(n)$



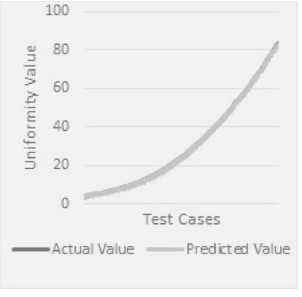
(o) T_1



(p) T_2



(q) V_n



(r) U_n^2

Fig. 1 (continued)

model–based metamorphic relation. Each bar was partitioned into the number of mutants that were killed (stripe fill) and the number of mutants that survived (solid fill).

Table 4 provides another lens on the data in Fig. 2. Figure 2 and Table 4 demonstrate that the regression model–based metamorphic relation was very effective for all of the subject programs, achieving an average mutation score of 92.96%. Even though there is some variation in the technique's effectiveness across the subject programs, the technique is relatively consistent. These results are promising and suggest that the regression model–based metamorphic relation is an effective and reliable approach to solving the oracle problem in implementations of uniformity statistics.

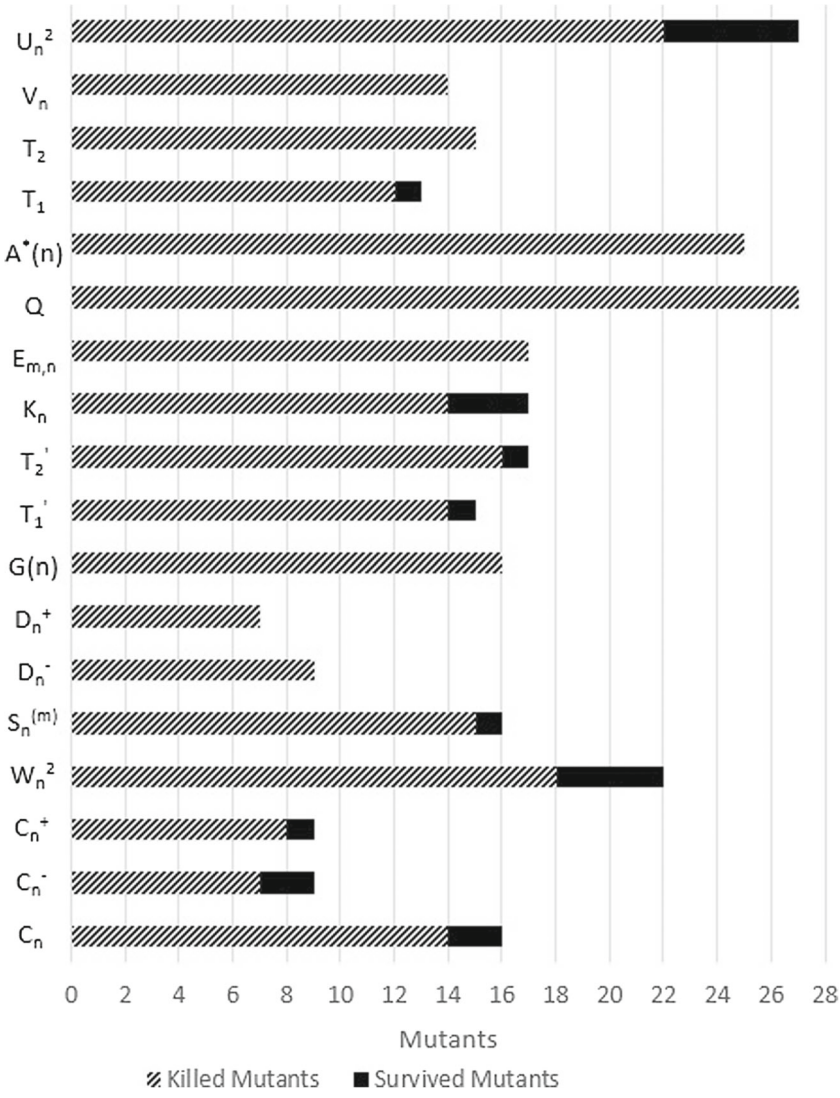


Fig. 2 Total number of mutants killed by the regression model–based metamorphic relation

5.4 RQ4. To what extent does the output of the mutant have to deviate from the output of the model in order for our technique to detect the difference?

The following procedure can be applied to a mutant, Mut. Let TS be a set of pairs $\langle \text{Output}_i, \text{Prediction}_i \rangle$, where Output_i is the output of Mut for a given test case tc_i and Prediction_i is the model's corresponding prediction. For each pair, $\langle \text{Output}_i, \text{Prediction}_i \rangle$, the absolute difference between Output_i and Prediction_i can be computed. Let AbsDiffs denote all of these absolute differences. The following tuple can be obtained by calculating the descriptive statistics of AbsDiffs: (Mean, StandardDeviation, Skewness, Kurtosis, Min, Max).

Table 4 Mutation score of each subject program

Subject program	MS	Subject program	MS	Subject program	MS
C_n	87.50%	D_n^+	100.00%	Q	100.00%
C_n^-	77.78%	$G(n)$	100.00%	$A^*(n)$	100.00%
C_n^+	88.89%	T_1'	93.33%	T_1	92.31%
W_n^2	81.82%	T_2'	94.12%	T_2	100.00%
$S_n^{(m)}$	93.75%	K_n	82.35%	V_n	100.00%
D_n^-	100.00%	$E_{m,n}$	100.00%	U_n^2	81.48%
(a)		(b)		(c)	

The aforementioned procedure can be applied to all of the killed mutants of a program, Prog, to obtain a set of tuples, $\text{Prog}_{\text{KilledTuples}}$, and applied to the original program to obtain one more tuple, $\text{Prog}_{\text{OriginalTuple}}$. The absolute difference between each tuple in $\text{Prog}_{\text{KilledTuples}}$ and $\text{Prog}_{\text{OriginalTuple}}$ can be computed. Let Mean_k, $\text{StandardDeviation}_k$, Skewness_k, Kurtosis_k, Min_k, and Max_k be the smallest absolute differences that were observed during this iterative process. Similarly, let $\text{Prog}_{\text{SurvivedTuples}}$ be the result of applying the aforementioned procedure to all of the survived mutants of Prog. The absolute difference between each tuple in $\text{Prog}_{\text{SurvivedTuples}}$ and $\text{Prog}_{\text{OriginalTuple}}$ can be computed. Let Mean_s, $\text{StandardDeviation}_s$, Skewness_s, Kurtosis_s, and Min_s, and Max_s be the largest absolute differences that were observed during this iterative process. Finally, the following boolean values can be calculated for Prog: $\text{Mean}_k \leq \text{Mean}_s$, $\text{StandardDeviation}_k \leq \text{StandardDeviation}_s$, $\text{Skewness}_k \leq \text{Skewness}_s$, $\text{Kurtosis}_k \leq \text{Kurtosis}_s$, $\text{Min}_k \leq \text{Min}_s$ and $\text{Max}_k \leq \text{Max}_s$. A boolean value of "True" indicates that there exists at least one killed mutant that is at least as close (if not closer) to the original program, than at least one survived mutant, based on the descriptive statistic that the boolean value corresponds to. Alternatively, a boolean value of "False" means that all of the survived mutants were closer to the original program than all of the killed mutants, based on the descriptive statistic that the boolean value corresponds to. Table 5 presents these boolean values for each subject program that did not have a mutation score of 100% and did not report a false positive.

Table 5 A comparison between killed and survived mutants

Subject program	Mean	Standard deviation	Skewness	Kurtosis	Min	Max
C_n	True	False	True	False	True	True
C_n^-	True	True	True	True	True	False
C_n^+	False	False	False	False	True	False
W_n^2	False	False	False	False	False	False
$S_n^{(m)}$	False	False	False	False	True	False
T_1'	False	False	False	False	False	True
K_n	True	True	True	True	True	True
T_1	False	False	False	False	True	False
U_n^2	True	True	True	True	True	True

Unsurprisingly, Table 5 shows that for most subject programs, all survived mutants were closer to the original program than the killed mutants were, based on at least one descriptive statistic. An interesting observation that can be made is that there exists two subject programs (K_n and U_n^2) in which the following holds true for each descriptive statistic: there exists at least one killed mutant that is at least as close (if not closer) to the original program than at least one survived mutant. Closer inspection of the raw data revealed that for both of these subject programs, there existed at least one killed mutant that was closer to the original program in terms of all descriptive statistics than at least one survived mutant.

Non-parametric statistics have been designed to be applicable in a wider range of scenarios than their parametric counterparts. However, the cost of their enhanced generalisability is a reduction in statistical power, and by implication non-parametric statistics are more likely to report that there is no effect, when there actually is. Since the Wilcoxon signed rank test is a non-parametric statistic, this might explain why some killed mutants were closer to the original program than some survived mutants.

5.5 RQ5. What impact would changing the alpha value have on the results?

Recall that an observation that was made in Section 5.2, was that one's choice of alpha value is an important determinant of the partial oracle's effectiveness. In this section, we explore this observation further.

A scatterplot in Fig. 3 plots the log of the p value (that was used for the pass/fail verdict) that was reported for each mutant (black markers) and the log of the p value (that was used for the pass/fail verdict) that was obtained for the original program (grey markers). The original program and mutants along the X-axis are organised into ascending order, based on the p value. Thus, black markers that are on the left-hand side of a grey marker represent mutants that acquired a lower p value than the original program, and black markers that are situated on the right-hand side of a grey marker pertain to mutants that have a higher p value than the original program. A logarithmic scale was used in these scatter plots for presentation purposes; in these scatterplots, it is assumed that $\log(0) = 0$.

Figure 3 can be used to conduct a sensitivity analysis as follows. One can superimpose a horizontal line at any point on the graph to represent the alpha value. With the exception of log(0), black markers that appear below the line represent killed mutants, and black markers that appear above the line represent survived mutants. Similarly, if the grey marker appears below the line, then there is a false positive, and if it appears above the line, then there is not a false positive.

A number of observations can be made from Fig. 3. For example, in every scatterplot, almost all, if not all mutants have a lower p value than the original program. This is promising because it demonstrates that the partial oracle can distinguish between mutants and the original program fairly accurately.

It can also be observed that the p value of the original program varies dramatically across subject programs. This means that choosing the optimal alpha value for a given subject program will be difficult. However, the results that were reported in Section 5.2 demonstrated that it is possible for the partial oracle to be effective in most cases with an alpha value of 0.000005, thus finding the optimal alpha value is not necessary if one opted to leverage this alpha value. It's also worth noting that in most cases, most mutants obtained a p value of 0, which means that changing the alpha to be arbitrarily close to 0 would have relatively little impact in most cases. Thus, using such an alpha might be a viable strategy.

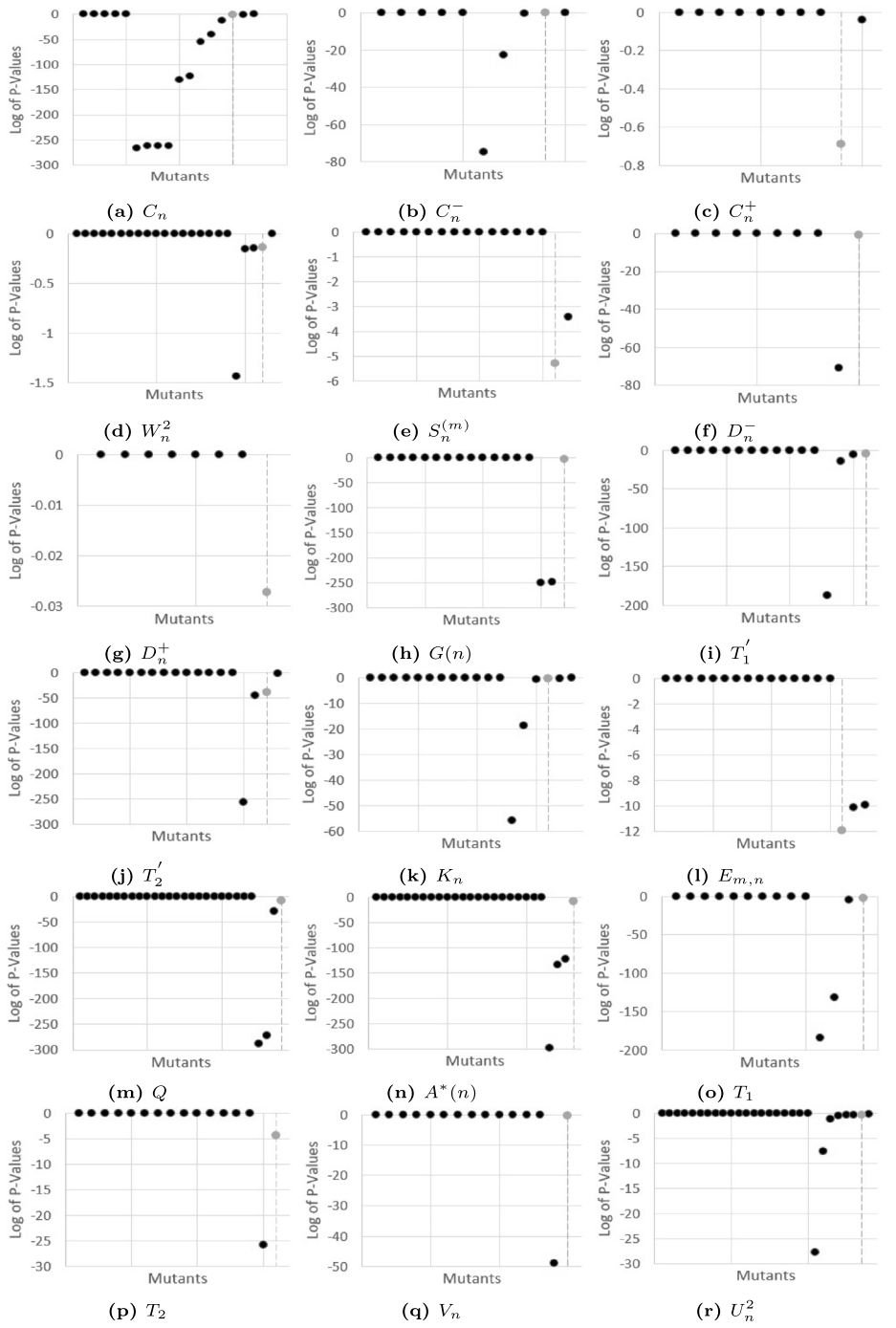


Fig. 3 A series of scatterplots that plot mutants against p values

5.6 RQ6. Why didn't the regression model-based metamorphic relation achieve a mutation score of 100%?

The regression model–based metamorphic relation was not able to kill all of the mutants. This section enumerates the key factors that were responsible for this outcome. Firstly, Sections 5.2 and 5.5 demonstrated that our choice of alpha value had an impact on the technique's ability (or lack thereof) to kill a mutant.

We conducted a similar analysis to the one that produced Table 5. This analysis differed in terms of its definitions of Mean_k, $\text{StandardDeviation}_k$, Skewness_k, Kurtosis_k, Min_k, Max_k. In particular, instead of being defined as the smallest absolute differences between $\text{Prog}_{\text{KilledTuples}}$ and $\text{Prog}_{\text{OriginalTuple}}$, they are defined as the average absolute differences between these tuples. Similarly, Mean_s, $\text{StandardDeviation}_s$, Skewness_s, Kurtosis_s, Min_s and Max_s are redefined to be the average absolute differences between $\text{Prog}_{\text{SurvivedTuples}}$ and $\text{Prog}_{\text{OriginalTuple}}$, instead of the largest absolute differences. Every cell in the newly produced table was false, which suggests that survived mutants are generally more similar to the original program than killed mutants. This indicates that the subtlety of mutants is another important factor.

Recall that some killed mutants were closer to the original program than some survived mutants (see Section 5.4). Also recall that one explanation for this is that our implementation of the technique incorporated the Wilcoxon signed rank test, which is a non-parametric statistic, and so is particularly susceptible to type 2 errors. This is another factor that could affect our technique's ability to kill mutants. The use of more robust statistics might alleviate this factor; we intend to explore the impact of using alternative statistics in future work.

5.7 Discussion

N-version testing is a testing technique, where the output that is produced by the program under test, in response to test case, tc, is compared with the output of a reference implementation of the program under test, for tc. N-version testing is a potential alternative to the regression model–based metamorphic relation; however, it has a number of limitations. For example, to the best of our knowledge, there are no readily available implementations available for a large number of the uniformity statistics that are covered in this paper. This necessitates the development of reference implementations, and thus could introduce correlated failures, i.e. the program under test and reference implementation might have the same faults and thus produce the same failures.

Since uniformity statistics operate on floating point numbers, it is necessary to define a similarity threshold. The relationship between standard deviation and uniformity is not linear in most cases. This means that an appropriate similarity threshold for one area of the output domain might not be appropriate for other regions of the output domain. This reduces the viability of applying N-version testing to uniformity statistics.

Since it can be difficult to obtain reference implementations, the option to use N-version testing might not be available. In such cases, it is possible for a domain expert to construct a training dataset for the regression model–based metamorphic relation or obtain this dataset through simulations. This means that the option to use regression model–based metamorphic relation can still be available in situations in which the option to use N-version testing is not.

6 Threats to validity

This section outlines the main threats to validity.

6.1 Internal validity

A number of tools were used to create our partial oracles, to carry out the experiments and to analyse the experimental results, e.g. SPSS (IBM 2015), R (Community 2017), Microsoft Excel and Mutmut (Hovmöller 2017). These tools might contain errors that could affect our results. However, all of these tools are widely used and highly reputable, and so it is unlikely that such faults exist. Additionally, such errors might also exist in our own code, e.g. subject programs, partial oracles, test case generators and analysis scripts. To reduce the likelihood of this, we thoroughly tested all of our code. Where possible, we also used automated tools to reduce the potential for bias to be introduced by the person performing the experiment.

6.2 External validity

There are three main threats to external validity. Firstly, even though steps were taken to ensure that a diverse range of uniformity statistics were included in our experiments (empirical distribution function, order, spacings and higher order spacing statistics (Marhuenda et al. 2005)), we did not include discrete uniformity statistics (Steele et al. 2005). Our results might have been different for such statistics; thus, this may threaten the generalisability of our results. We will seek to address this in future work.

Secondly, metamorphic testing is often conducted with several metamorphic relations, but in this paper, we only leveraged one metamorphic relation. Our reasoning is twofold. Firstly, we were unable to identify any more general metamorphic relations. Even though it might have been possible to construct metamorphic relations that would be useful for individual subject programs, such metamorphic relations would be generally less useful.

In our experiments, we only leveraged test cases that were generated from a normal distribution. Our results might have been different, had we drawn test cases from a different distribution, e.g. a bimodal distribution. We intend to investigate the impact of changing the distribution, on our technique, in future work.

Finally, all of the subject programs that were used in our experiments were developed by the authors. This is a threat to validity, because our implementations might not be representative of third-party implementations. We attempted to alleviate this threat by including third-party implementations in our experiments, but we were unable to find any suitable implementations. We therefore decided to compare the functionality of some of our implementations with corresponding R code. In order to do so, we randomly generated 100 test cases and evaluated stat0064, stat0067, stat0068, stat0069, stat0070, stat0071 and stat0077 from R's PoweR package (see the documentation on R's PowerR package for more details), as well as our implementations of these statistics. We then compared the outputs that were produced by our implementations and R's implementations, using the Mann-Whitney U test. None of the tests produced a significant result at an alpha value of 0.05. This suggests that our implementations are representative of third-party implementations.

6.3 Construct validity

The two main metrics used in this paper are the mutation score and false positive rate. These metrics are widely used by the research community. The first of these metrics is

used so frequently by the research community that some consider them to be a de-facto standard (Segura et al. 2016).

6.4 Statistical validity

We used a number of statistical tests, including Fisher's exact test and the Wilcoxon signed rank test (Pallant 2007) in this paper. These statistical tests are non-parametric and were used partly because their assumptions had been fulfilled by the data, and also because their parametric counterparts were not applicable. Given that these statistics were applied a large number of times in certain analyses, each of these analyses had a high probability of falsely reporting at least one significant difference. To that end, we applied Benjamini-Hochberg correction to these analyses to account for this.

7 Conclusions

Implementations of uniformity statistics suffer from the oracle problem because it is infeasible to predict the test statistic value of a given sample. In this paper, we attempted to address this by formulating a partial oracle called the regression model--based metamorphic relation.

We also investigated the effectiveness of the regression model–based metamorphic relation. We discovered that, when appropriate p values are used, the technique can achieve mutation scores ranging from 77.78 to 100% (tending towards higher mutation scores), and produce no false alarms. Our experiments also revealed that the models that the regression model–based metamorphic relation are based on are incredibly accurate. These results are promising and suggest that the regression model–based metamorphic relation is a viable means of ameliorating the oracle problem in the implementation of uniformity statistics.

The predictive models used by the regression model–based metamorphic relation were primarily created based on the relationship between the test statistic value and standard deviation. There might be scope to incorporate other features of the sample, instead of just the standard deviation, to improve their accuracy further; improvements in accuracy would allow us to tune the partial oracles to be more sensitive. Thus, one avenue of future work might include investigating the viability of these methods.

The regression model–based metamorphic relation is dependent on predictive models that are learned from a dataset. In practice, one could obtain such a dataset from the SUT, but models that are derived from such a dataset would only be suitable for regression testing. If the tester wishes to apply these partial oracles in a more general context, they would either have to acquire such a dataset from elsewhere, e.g. from a reference implementation, or request a domain expert to create the model based on their knowledge of the system. Unfortunately, a scenario may exist in which neither a reference implementation, nor a domain expert are available. In such situations, these partial oracles are rendered inapplicable. This is an important limitation of the approaches, and one that we wish to address in future work.

We believe that implementations of other classes of statistics, e.g. correlation statistics, might have similar oracle problems. The partial oracles that were presented in this paper are only suitable for uniformity statistics. Thus, future work that explores methods of alleviating the oracle problem for implementations of other classes of statistics may be invaluable.

Funding information This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P006116/1, InfoTestSS: Information Theory and Test Suite Selection.

Appendix : Uniformity statistics

Algorithm 2 D_n^+.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: D_n^+

Let $Results$ be an empty set;
for $U_{(i)}$ *in* $Sample_o$ **do**
    $Result = (i/n) - U_{(i)}$;
    $Results.add(Result)$;
end
D_n^+ is the member of $Results$ that has the highest value;

Algorithm 3 D_n^-.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: D_n^-

Let $Results$ be an empty set;
for $U_{(i)}$ *in* $Sample_o$ **do**
    $Result = U_{(i)} - ((i-1)/n)$;
    $Results.add(Result)$;
end
D_n^- is the member of $Results$ that has the highest value;

Algorithm 4 V_n.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: V_n

V_n is the result of adding D_n^+ to D_n^-;

Algorithm 5 W_n^2.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: W_n^2

$W_n^2 = 0$;
for $U_{(i)}$ *in* $Sample_o$ **do**
  $W_n^2 = W_n^2 + (U_{(i)} - (((2 * i) - 1)/(2 * n)))^2$;
end
$W_n^2 = W_n^2 + (1/(12 * n))$;

Algorithm 6 U_n^2.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: U_n^2

Let W_n^2 be the result of computing the W_n^2 statistic;
$U_n^2 = W_n^2 - (n * (\bar{U} - 0.5)^2)$, where $\bar{U}$ is the average (mean) of the sample;

Algorithm 7 C_n^+.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: C_n^+

$SetOfV = \emptyset$;
for $i = 1$ *to* n **do**
  $m_i = i/(n + 1)$;
  $v_i = U_{(i)} - m_i$;
  $SetOfV.add(v_i)$;
end
C_n^+ is the largest member of $SetOfV$;

Algorithm 8 C_n^-.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: C_n^-

$SetOfV = \emptyset$;
for $i = 1$ *to* n **do**
  $m_i = i/(n + 1)$;
  $v_i = U_{(i)} - m_i$;
  $SetOfV.add(-v_i)$;
end
C_n^- is the largest member of $SetOfV$;

Algorithm 9 C_n.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: C_n

Compute C_n^+ and C_n^-;
if $C_n^+ > C_n^-$ **then**
    $C_n = C_n^+$;
else if $C_n^+ < C_n^-$ **then**
    $C_n = C_n^-$;

Algorithm 10 K_n.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: K_n

Compute C_n^+ and C_n^-;
K_n is the result of adding C_n^+ to C_n^-;

Algorithm 11 T_1.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: T_1

$T_1 = 0$;
for $i = 1$ *to* n **do**
    $m_i = i/(n+1)$;
    $v_i = U_{(i)} - m_i$;
    $T_1 = T_1 + (Math.abs(v_i)/n)$;
end

Algorithm 12 T_2.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: T_2

$T_2 = 0$;
for $i = 1$ *to* n **do**
    $m_i = i/(n+1)$;
    $v_i = U_{(i)} - m_i$;
    $T_2 = T_2 + (v_i^2/n)$;
end

Algorithm 13 T_1'.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: T_1'

$T_1' = 0$;
for $i = 1$ *to* n **do**
  $v_i = U_{(i)} - (i-1)/(n-1)$;
  $T_1' = T_1' + (Math.abs(v_i)/n)$;
end

Algorithm 14 T_2'.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: T_2'

$T_2' = 0$;
for $i = 1$ *to* n **do**
  $v_i = U_{(i)} - (i-1)/(n-1)$;
  $T_2' = T_2' + (v_i^2/n)$;
end

Algorithm 15 $G(n)$.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: $G(n)$

$G(n) = 0$;
for $i = 1$ *to* $n+1$ **do**
  if $i = 1$ **then**
    $D_i = U_{(i)}$;
  else if $i = n+1$ **then**
    $D_i = 1 - U_{(n)}$;
  else
    $D_i = U_{(i)} - U_{(i-1)}$
  end
  $G(n) = G(n) + D_i^2$;
end

Algorithm 16 Q.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: Q

$Spacings = emptylist$;
for $i = 1$ *to* $n + 1$ **do**
  if $i = 1$ **then**
    $D_i = U_{(i)}$;
  else if $i = n + 1$ **then**
    $D_i = 1 - U_{(n)}$;
  else
    $D_i = U_{(i)} - U_{(i-1)}$;
  end
  $Spacings.add(D_i)$;
end
$Q = 0$;
for $i = 1$ *to* n **do**
  $Q = Q + Spacings.D_i * Spacings.D_{i+1}$;
end
Let $G(n)$ be the result of applying statistic $G(n)$;
$Q = Q + G(n)$;

Algorithm 17 $S_n^{(m)}$.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.). m: a constant integer selected by the user to determine the gap between elements.

Output: $S_n^{(m)}$

$S_n^{(m)} = 0$;
for $i = 0$ *to* $n + 1 - m$ **do**
  $G_i^{(m)} = U_{(i+m)} - U_{(i)}$, where m denotes a constant integer selected by the user to determine the gap between elements. In this calculation, $U_{(0)} = 0$ and $U_{(n+1)} = 1$;
  $S_n^{(m)} = S_n^{(m)} + (n * G_i^{(m)})^2$;
end

Algorithm 18 $A^*(n)$.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.)

Output: $A^*(n)$

$A^*(n) = 0$;
for $i = 1$ *to* n **do**
  $A^*(n) = A^*(n) + (((U_{(i+1)} - U_{(i-1)})/2) - 1/n)^2$. In this equation, $U_{(0)} = -U(1)$ and $U_{(n+1)} = 2 - U_{(n)}$;
end
$A^*(n) = A^*(n) * (n/2)$;

Algorithm 19 $E_{m,n}$.

Input: $Sample_o$: an ordered sample $U_{(1)}, U_{(2)},...,U_{(n)}$ (sorted from smallest to largest, $U_{(i)}$ denotes an element in the sample, i denotes the position of element $U_{(i)}$ in the sample, and n is the sample size.). m: a constant integer selected by the user to determine the gap between elements.

Output: $E_{m,n}$

$E_{m,n} = 0$;
for $i = 1$ *to* n **do**
  $E_{m,n} = E_{m,n} + (2 * m)/(n * (U_{(i+m)} - U_{(i-m)}))$. For this equation, $U_{(i)} = U_{(1)}$ if $i < 1$, and $U_{(i)} = U_{(n)}$ if $i > n$;
end
$E_{m,n} = E_{m,n} * (1/n)$;

References

Andrews, J.H., Briand, L.C., Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on software engineering (ICSE)* (pp. 402–411). Saint Louis: ACM.

Androutsopoulos, K., Clark, D., Dan, H., Hierons, R.M., Harman, M. (2014). An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th international conference on software engineering* (pp. 573–583). New York: ACM.

Cao, Y., Zhou, Z.Q., Chen, T.Y. (2013). On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the 13th international conference on quality software (QSIC)* (pp. 153–162). Najing: IEEE.

Chan, W.K., Cheung, S.C., Ho, J.C.F., Tse, T.H. (2006). Reference models and automatic oracles for the testing of mesh simplification software for graphics rendering. In *Proceedings of the 30th annual international computer software and applications conference (COMPSAC)* (pp. 429–438). Chicago: IEEE.

Chan, W.K., Cheung, S.C., Leung, K.R.P.H. (2007). A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, *4*(2), 61–81.

Chan, W.K., Ho, J.C.F., Tse, T.H. (2010). Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, *20*(2), 89–120.

Chen, T. (2013). Studying software quality using topic models. Technical report, Queen's University, Kingston, Ontario, Canada.

Chen, T.Y., Cheung, S.C., Yiu, S.M. (1998). Metamorphic testing: a new approach for generating next test cases. Technical report HKUST-CS98-01 Hong Kong University of Science and Technology.

Chen, T.Y., Ho, J.W.K., Liu, H., Xie, X. (2009). An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, *10*(1), 1–12.

Chen, T.Y., Kuo, F.-C., Liu, H., Poon, P.-L., Towey, D., Tse, T.H., Zhou, Z.Q. (2018). Metamorphic testing: a review of challenges and opportunities. *ACM Computing Surveys*, *51*(1), 4:1–4:27.

Chen, T.Y., Kuo, F.-C., Towey, D., Zhou, Z.Q. (2015). A revisit of three studies related to random testing. *SCIENCE CHINA Information Sciences*, *58*(5), 1–9.

Chen, T.Y., Leung, H., Mak, I.K. (2004). Adaptive random testing. volume 3321 of Lecture Notes in Computer Science, pp. 320–329, Springer.

Chen, T.Y., Tse, T.H., Zhou, Z.Q. (2003). Fault-based testing without the need of oracles. *Information and Software Technology*, *45*(1), 1–9.

Chockler, H., Farchi, E., Godlin, B., Novikov, S. (2007). Cross-entropy based testing. In *Formal methods in computer aided design* (pp. 101–108). Texas: IEEE.

Claessen, K., Duregard, J., Pałka, M.H. (2014). Generating constrained random data with uniform distribution. In *International symposium on functional and logic programming, Kanazawa, Japan* (pp. 18–34). Cham: Springer.

Community, R. (2017). The R project for statistical computing. https://www.r-project.org.

Dutra, R., Laeufer, K., Bachrach, J., Sen, K. (2018). Efficient sampling of sat solutions for testing. In *Proceedings of the 40th international conference on software engineering* (pp. 1–11). Gothenburg: ACM.

Feldt, R., Torkar, R., Gorschek, T., Afzal, W. (2008). Searching for cognitively diverse tests: towards universal test diversity metrics. In *First international conference on software testing verification and validation (ICST 2008)* (pp. 178–186): IEEE Computer Society.

Guderlei, R., & Mayer, J. (2007). Statistical metamorphic testing – testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proceedings of the 7th international conference on quality software (QSIC)* (pp. 404–409). Oregon: IEEE.

Harrison, R.L. (2010). Introduction to Monte Carlo simulation. In *AIP conference proceedings, USA, National Institute of Health* (pp. 1–14).

Hovmöller, A. (2017). Mutmut. https://pypi.python.org/pypi/mutmut.

Hummel, O., Atkinson, C., Brenner, D., Keklik, S. (2006). Improving testing efficiency through component harvesting. Technical report, University of Mannheim.

IBM (2015). IBM SPSS statistics. https://www.ibm.com/uk-en/marketplace/spss-statistics.

Kanewala, U., & Bieman, J.M. (2013). Techniques for testing scientific programs without an oracle. In *Proceedings of the 5th international workshop on software engineering for computational science and engineering (SE-CSE)* (pp. 48–57). California: IEEE.

Kanewala, U., & Bieman, J.M. (2015). Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software Testing, Verification, and Reliability*, *26*(3), 245–269.

Liu, H., Kuo, F.-C., Towey, D., Chen, T.Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, *40*(1), 4–22.

Liu, H., Xie, X., Yang, J., Lu, Y., Chen, T.Y. (2010). Adaptive random testing by exclusion through test profile. In *Proceedings of the 10th international conference on quality software* (pp. 147–156). Zhangjiajie: IEEE.

Liu, H., Yusuf, I.I., Schmidt, H.W., Chen, T.Y. (2014). Metamorphic fault tolerance: an automated and systematic methodology for fault tolerance in the absence of test oracle. In *Proceedings of the 36th international conference on software engineering* (pp. 420–423). New York: ACM.

Marhuenda, Y., Morales, D., Pardo, M.C. (2005). A comparison of uniformity tests. *Statistics*, *39*(4), 315–328.

Mayer, J. (2005). On testing image processing applications with statistical methods. In *Proceedings of software engineering, Bonn, Germany* (pp. 69–78). Lecture Notes in Informatics.

McMinn, P. (2009). Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the 11th annual conference on genetic and evolutionary computation* (pp. 1689–1696). New York: ACM.

Offutt, J.A. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, *1*(1), 5–20.

Pallant, J. (2007). SPSS survival manual: a step-by-step guide to data analysis using SPSS version 15. Open University Press.

Patel, K., & Hierons, R.M. (2017). A mapping study on testing non-testable systems. *Software Quality Journal*, *26*(4), 1–41.

Segura, S., Fraser, G., Sánchez, A.B., Cortés, A.R. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, *42*(9), 805–824.

Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortes, A. (2016). Metamorphic testing: a literature review. Technical report ISA-16-TR-02, applied software engineering research group, University of Seville, Spain.

Steele, M., Chaseling, J., Hurst, C. (2005). Simulated power of the discrete cramer-von mises goodness-of-fit tests. In *Proceedings of the international congress on modelling and simulation (MODSIM)* (pp. 1300–1304). Melbourne: MSSANZ.

Weyuker, E.J. (1982). On testing non-testable programs. *The Computer Journal*, *25*(4), 465–470.

Wu, P., Shi, X.-C., Tang, J.-J., Lin, H.-M., Chen, T.Y. (2005). Metamorphic testing and special case testing: a case study. *Journal of Software*, *16*(7), 1210–1220.

Zhang, Z., Chan, W.K., Tse, T.H., Hu, P. (2009). Experimental study to compare the use of metamorphic testing and assertion checking. *Journal of Software*, *20*(10), 2637–2654.

Zhou, Z.Q., Huang, D.H., Tse, T.H., Yang, Z., Huang, H., Chen, T.Y. (2004). *Metamorphic testing and its applications*, (pp. 1–6). Xian: The Software Engineers Association.

Krishna Patel received a BSc and Ph.D. in Computer Science at Brunel University London. He then served as a Research Fellow at Brunel University London between 2017 and 2018, where he was a member of the Brunel Software Engineering Lab (BSEL), before joining the Testing Research Group at the University of Sheffield in 2018.

Robert M. Hierons received a BA in Mathematics (Trinity College, Cambridge), and a Ph.D. in Computer Science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full Professor in 2003 and joined The University of Sheffield in 2018.