An Empirical Study on the Selection of Good Metamorphic Relations

Johannes Mayer*
Ulm University
Dept. of Applied Information Processing
89069 Ulm, Germany
Email: johannes.mayer@uni-ulm.de

Ralph Guderlei†
Ulm University
Dept. of Applied Information Processing
89069 Ulm, Germany
Email: ralph.guderlei@uni-ulm.de

Abstract

Full software test automation requires automated test input generation, execution, and output evaluation. The latter task is non-trivial and usually referred to as the oracle problem in software testing. The present paper describes an empirical study on Metamorphic Testing, an approach to the oracle problem. This study was conducted with common Java implementations of determinant computation in order to evaluate the usefulness of the Metamorphic Testing approach and to establish general criteria that can be used to quickly assess metamorphic relations with respect to their suitability. The latter is very important, since Metamorphic Testing is based on so-called metamorphic relations on input-output tuples, which can easily be found. It is, however, crucial to evaluate these relations according to their usefulness. The empirical study enables us to derive general rules that can be used to quickly assess metamorphic relations and identify those that should be considered and studied in more detail with other methods (e. g. with mutation analysis).

1. Introduction

Often, automated test execution is understood as automated testing. This is, however, only the first step towards test automation. More important is the automated generation of test input data and, especially, the automatic evaluation of the test outputs, which results in a pass/fail decision for the test case. The second problem is often referred to as the *oracle problem* [36].

It is not reasonable to assume that a perfect oracle exists, i. e. one that can evaluate each test output correctly. There are even "non-testable" programs discussed in [36]. Therefore, general oracle solutions cannot be found. Oracles presented in the literature so far are only applicable to special cases. Approaches to the oracle problem are described e. g. in [4] and [22]. A weaker version of the perfect oracle is the so-called *Gold Standard Oracle* [4]. It consists of a trusted implementation, e. g. a legacy system, which can be used in parallel to the system under test (SUT) to obtain the reference values. There are oracles that only check some characteristics of the output or can only decide a small subset of all possible test input data, such as the *Heuristic Oracle* [22]. Many other approaches to the oracle problem exist that either cannot be automated or are only applicable to a small set of problems (such as e. g. the *Reversing Oracle* [22]). If no golden implementation is available, another oracle has to be found for test automation. Blum et al. [5, 6] introduced the concept of program checkers as probabilistic algorithms that can evaluate the correctness of the outcome of a computation with adjustable error probability. These probabilistic oracle algorithms have been studied mostly in theoretical computer science.

Chen et al. [7, 9, 12] introduced *Metamorphic Testing* an approach that checks necessary properties of tuples of outputs whose inputs also satisfy a certain property. Such a property of input and output tuples is called a *metamorphic relation*. These relations are not difficult to obtain, but not all of them are useful. The evaluation of these relations with respect to their suitability is therefore crucial.

In the present paper, we describe an empirical study that was designed to evaluate the usefulness of Metamorphic Testing and to establish criteria for good metamorphic relations. This study is based on common Java implementations of determinant computation.

In Section 2 the Metamorphic Testing approach is introduced and its applications are described. An empirical study of Metamorphic Testing applied to determinant computation implemented in Java is described in Section 3, followed by a discussion of the results of this study. Section 4 presents general criteria derived from the study which allow to quickly assess metamorphic relation with respect to their

*Corresponding author.

†Work supported by the Deutsche Forschungsgemeinschaft in the research training group "Modellierung, Analyse und Simulation in der Wirtschaftsmathematik"

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

suitability. Section 5 concludes the paper.

2. Metamorphic Testing

Chen et al. [7, 9, 12] proposed an approach called *Metamorphic Testing*, which is more general than the Heuristic Oracle [22] and applicable to arbitrary inputs. This approach partially combines input generation (of follow-up test cases) and output evaluation. In this paradigm, existing relations on input and output values are employed. For a given tuple of input data $(i_1, \ldots i_n)$, $n \geq 1$, the SUT is supposed to compute a function f, i. e. $f(i_1), \ldots, f(i_n)$ are the expected outputs. The function f can be regarded as the formal specification of the SUT. Furthermore, let R and R_f be two relations over tuples of inputs and outputs, respectively, satisfying

$$R(i_1, \ldots, i_n) \Rightarrow R_f(f(i_1), \ldots, f(i_n))$$

for each tuple $(i_1, \ldots i_n)$ of inputs. Then, each implementation I of f has to satisfy

$$R(i_1, \ldots, i_n) \Rightarrow R_f(I(i_1), \ldots, I(i_n))$$

for each tuple $(i_1, \ldots i_n)$ of inputs. The latter property is called a *metamorphic relation*. Note that it is only a necessary condition for the correctness of I with respect to f. Furthermore, one can only test the metamorphic relation above for finitely many tuples $(i_1, \ldots i_n)$ of inputs—the usual restriction of testing. Therefore, one cannot be sure that it holds for all inputs. Obviously, Metamorphic Testing has to be complemented by a meaningful strategy to select the test inputs also considering special and boundary cases.

It is quite common for metamorphic relations that some $i_1, \ldots, i_k$ can be chosen arbitrarily. The so-called *follow-up test cases* $i_{k+1}, \ldots, i_n$ are then determined by $i_1, \ldots, i_k$ and have to be chosen such that $R(i_1, \ldots, i_n)$ holds. The test cases $i_1, \ldots, i_k$ are chosen by an arbitrary strategy for test input generation and the follow-up test cases have to be added (according to the metamorphic relation which is used).

Often, symmetry properties can be used as metamorphic relations, such as

$$gcd(a, b) = gcd(b, a),$$

$a, b \in \mathbb{Z}, a^2 + b^2 > 0$, for the greatest common divisor (gcd). This special case is described in [19] as Symmetric Testing where algorithmic advantages obtained by group theory are described. For an arbitrarily chosen test case (a, b), a follow-up test case (b, a) can be generated according to the above metamorphic relation. Thereafter, both test cases are evaluated using the relation.

The big challenge related to Metamorphic Testing is the identification of suitable metamorphic relations. It is simple to identify arbitrary metamorphic relations, such as the above, but difficult to evaluate their suitability. A better metamorphic relation than the above one for implementations of the greatest common divisor is

$$k \cdot gcd(a, b) = gcd(k \cdot a, k \cdot b)$$

for each $a, b \in \mathbb{Z}, a^2 + b^2 > 0$, $k \in \mathbb{N}_+$, since it describes much of the semantics of gcd.

Metamorphic Testing has been applied successfully in many cases. In [38], an overview of Metamorphic Testing and possible applications in areas such as numerics, graph theory, computer graphics, compilers, and interactive software is given. In [10] it has been applied to test implementations of programs that solve partial differential equations. An application of Metamorphic Testing to context-sensitive middleware applications is described in [34, 8]. A case study is described in [11], where Metamorphic Testing has been used to test implementations of shortest and critical path algorithms. Some hints regarding the selection of metamorphic relations have been devised.

A similar approach to Metamorphic Testing is described in [1] in order to obtain fault-tolerant software, where the outputs are compared for equality, i. e. $R_f = \{(f_1, \ldots, f_n) \mid f_1 = \ldots = f_n\}$.

In [20], a testing framework is introduced that is able to find test inputs that violate a given metamorphic relation. For this purpose, Constraint Logic Programming is used. Under certain circumstances, this framework even proves that an implementation fulfills a given metamorphic relation.

In semi-proving [13], it is verified using symbolic inputs that a program satisfies a metamorphic relation (at least for an execution path). This technique combines metamorphic testing and formal verification.

The tool QuickCheck implemented in Haskell is described in [14]. It enables programmers to specify properties in Haskell that are automatically tested with randomly generated inputs. Programmers can also implement other input generators. The properties used by QuickCheck are equivalent to metamorphic relations.

Distributional properties have been used in [25, 26, 27] as metamorphic "relations" to tests image processing and analysis applications.

3. Empirical Study

In the following an empirical study with common Java implementations of determinant computation is described. After the introduction of relevant basics, metamorphic relations are given that can be used for determinant computation. Thereafter, a description of the study scenario is given, and the results are presented and discussed. From these results, general criteria for the assessment of metamorphic relations will be deduced in the following section.

IEEE COMPUTER SOCIETY

3.1. Preliminaries

The following description of matrices and determinants is based on [28] where further details can be found. $\mathbb{R}_{n,m}$ is the set of all matrices with real elements, n rows, and m columns. A matrix $A \in \mathbb{R}_{n,m}$ with

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{pmatrix}$$

can also be expressed by its column vectors $A = (a^1, \ldots, a^m)$ resp. its row vectors $A = (a_1, \ldots, a_n)^T$, where A^T denotes the *transposition* of matrix A, i. e. the matrix with columns and rows being exchanged.

For a matrix $A = (a_{i,j})_{1 \le i,j \le n} \in \mathbb{R}_{n,n}$ let $det(A) = |A|$ denote the *determinant* of A which is defined as the scalar

$$det(A) = \begin{vmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{vmatrix} := \sum_{\pi \in S_n} sgn(\pi) \prod_{j=1}^{n} a_{j,\pi(j)}$$

where S_n is the permutation group on $\{1, \ldots, n\}$, i. e. S_n contains all permutations on $\{1, \ldots, n\}$, and $sgn(\pi)$ denotes the *sign* of the permutation π. The sign of a permutation can be 1 or -1. It is 1 if an even number of swaps (of two elements) is necessary to generate it from the identity permutation, and -1 otherwise.

Let A be a matrix from $\mathbb{R}_{n,n}$. A is said to be *singular* if and only if there exist constants $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$, $\sum_{i=1}^{n} |\lambda_i| > 0$ with $\sum_{i=1}^{n} \lambda_i a_i = o$, where o denotes the origin of $\mathbb{R}_n$, i. e. $o = (0, \ldots, 0)$. A non-singular matrix is called *regular*. Thus, a random matrix will almost certainly be regular, since constants λ_i with $\lambda_k \neq 0$ for some k have to exist such that $a_k = -\frac{1}{\lambda_k} \sum_{1 \le i \le n, i \neq k} \lambda_i a_i$ if A was singular. This is, however, only possible with probability 0 if the elements of A are uniformly distributed over $\mathbb{R}$ or $[b, c] \subset \mathbb{R}$, $b < c$. Furthermore, $det(A) \neq 0$ if and only if A is a *regular* matrix.

The *submatrix* $A_{i,j}$ of the matrix $A \in \mathbb{R}_{n,m}$ is obtained by deleting the i th row and the j th column from A. The determinant $det(A_{i,j})$ of a submatrix $A_{i,j} \in \mathbb{R}_{n-1,n-1}$ of $A \in \mathbb{R}_{n,m}$ is called a *minor* of A.

3.2. Metamorphic Relations for Determinant Computations

There are many relations for determinants that can be used as metamorphic relations (cf. [28]). In the following, we assume that all matrices A, B are chosen from $\mathbb{R}_{n,n}$. No further assumptions are necessary. Consequently, the matrices can be chosen deterministically or randomly.

R1 *Transposition*
$det(A) = det(A^T)$

R2 *Exchange of Rows*
$-\, det((a_1, \ldots, a_n)^T)$
$= det((a_1, \ldots, a_{j-1}, a_i, a_{j+1}, \ldots, a_{i-1}, a_j, a_{i+1}, \ldots, a_n)^T)$
for $i, j \in \{1, \ldots, n\}, j < i$

R3 *Exchange of Columns*
$-\, det((a^1, \ldots, a^n))$
$= det((a^1, \ldots, a^{j-1}, a^i, a^{j+1}, \ldots, a^{i-1}, a^j, a^{i+1}, \ldots, a^n))$
for $i, j \in \{1, \ldots, n\}, j < i$

R4 *Row Multiplied with Scalar*
$\beta\, det((a_1, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n)^T)$
$= det((a_1, \ldots, a_{k-1}, \beta a_k, a_{k+1}, \ldots, a_n)^T)$
for $k \in \{1, \ldots, n\}$

R5 *Column Multiplied with Scalar*
$\beta\, det((a^1, \ldots, a^{k-1}, a^k, a^{k+1}, \ldots, a^n))$
$= det((a^1, \ldots, a^{k-1}, \beta a^k, a^{k+1}, \ldots, a^n))$
for $k \in \{1, \ldots, n\}$

R6 *Addition of Rows of Two Matrices*
$det((a_1, \ldots, a_{k-1}, a_k + b_k, a_{k+1}, \ldots, a_n)^T)$
$= det((a_1, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n)^T)$
$+\, det((a_1, \ldots, a_{k-1}, b_k, a_{k+1}, \ldots, a_n)^T)$
for $k \in \{1, \ldots, n\}$

R7 *Addition of Columns of Two Matrices*
$det((a^1, \ldots, a^{k-1}, a^k + b^k, a^{k+1}, \ldots, a^n))$
$= det((a^1, \ldots, a^{k-1}, a^k, a^{k+1}, \ldots, a^n))$
$+\, det((a^1, \ldots, a^{k-1}, b^k, a^{k+1}, \ldots, a^n))$
for $k \in \{1, \ldots, n\}$

R8 *Addition of Rows*
$det((a_1, \ldots, a_n)^T)$
$= det((a_1, \ldots, a_{j-1}, a_j + \beta a_i, a_{j+1}, \ldots, a_n)^T)$
for $\beta \in \mathbb{R}, i, j \in \{1, \ldots, n\}, i \neq j$

R9 *Addition of Columns*
$det((a^1, \ldots, a^n))$
$= det((a^1, \ldots, a^{j-1}, a^j + \beta a^i, a^{j+1}, \ldots, a^n))$
for $\beta \in \mathbb{R}, i \in \{1, \ldots, n\}, i \neq j$

R10 *Cofactor Expansion of a Row*
$det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j}\, det(A_{i,j})$
for $i \in \{1, \ldots, n\}$

R11 *Cofactor Expansion of a Column*
$det(A) = \sum_{i=1}^{n} (-1)^{i+j} a_{i,j}\, det(A_{i,j})$
for $j \in \{1, \ldots, n\}$

R12 *Multiplication*
$det(A) \cdot det(B) = det(AB)$

IEEE COMPUTER SOCIETY

The above metamorphic relations are not in the notation used in the description of Metamorphic Testing. However, it is simple to transform them. For example, if the inputs (A, B) satisfy $B = A^T$, it follows $det(A) = det(B)$ (according to R1). Therefore, A can be chosen according to an arbitrary strategy for R1 and the follow-up test case $B = A^T$ is then determined (by the input relation). For R6, e. g., A, b_k, and k can be chosen arbitrarily.

In addition to the previously defined metamorphic relations, four combinations of these relations were used. It was our aim to improve the fault detection capabilities of single relations by combining structural different relations. Relations $R13$ and $R14$ combine $R10$, $R12$, and $R11$, $R12$, respectively. Relation $R15$ is a combination of $R10$, $R11$, and $R12$. The fourth combination of relations is $R16$ which combines $R2$, $R3$, $R10$, $R11$, and $R12$. As Different metamorpic relations check different aspects of the SUT, the basic idea of the combinations is to improve the fault detection capabilities of the test cases induced by the metamorpic relations. The combinations presented above are only exemplary. To combine the metamorpic relations, the same input values were used by different metamorpic relations to evaluate the SUT. An error is revealed if one of the metamorphic relations does not hold.

Although the metamorphic relations could be used for deterministic inputs chosen according to an arbitrary strategy, random generation will be used in the following to enable simple automated testing.

3.3. Classification of Possible Test Inputs

The input domain can be divided into two major parts: square and non-square matrices. As determinants are defined for square matrices only, all implementations should indicate an error in case of non-square matrices.

The following cases should be regarded for square matrices:

1. Matrix with without elements ($A \in \mathbb{R}_{0,0}$): This is a degenerated case but nevertheless it has to be tested. As the computation of the determinant hardly makes sense in this case, it is expected that the SUT indicates an error.

2. Matrices with one element ($A \in \mathbb{R}_{1,1}$): In this cases, the determinant is equal to the only element.

3. Irregular matrices: The determinant is always equal to 0 in this case.

4. Regular matrices with more than one element: In this case, the computation of the determinant is a non-trivial task in general.

Only the latter two cases will be considered in the following, since metamorphic relations are necessary for test evaluation only in these cases. (Keep in mind that irregular matrices cannot be detected easily.) For the other cases, a perfect oracle is available.

3.4. Study/Scenario

For the study, six common Java implementations for the computation of matrix determinants were chosen. All of them are available in source code. The following implementations were examined:

- Commons.Math from the Apache Jakarta project, version 1.0 [3]
- JScience, version 2.0.1 [15]
- JAMA, version 1.0.2 [23]
- an implementation of Michael Thomas Flanagan, version of 2005/05/01 [17]
- an implementation of Jon Squire, downloaded on 2005/10/20 [33], and
- the GeoStoch library of Ulm University [35].

All implementations compute the determinant using LU decomposition. For the computation of the LU decomposition, a Gaussian elimination with partial pivoting is used.

To measure the effectiveness of the proposed method, the so-called *Mutation Testing* technique [16, 31] was used. Mutation Testing is a fault-based method to examine the quality of a given test set. Therefore, small faults are introduced to the original implementation using so-called *mutation operators* [24]. The resulting faulty version of the implementation is called a *mutant*. Then, the test cases are passed to the original implementation and to the mutants. If the outputs differ, the fault in the mutant is revealed, the mutant is said to be killed. In [29] it is shown that simple mutations are sufficient as test cases which reveal simple faults also reveal more complex faults. The efficiency of the test set can be measured using the so-called mutation score, the number of killed mutants divided by the total number of mutants (equivalent mutants are excluded). In [2] it is shown that Mutation Testing can also be used to examine the efficiency of a testing technique, as in this case, Metamorphic Testing.

We used the tool MuJava [30] to automatically generate the mutants. Table 1 gives an overview of the number of mutants. It shows the total number of generated mutants in the fifth column and the number of normally terminating, exception-throwing, and non-terminating mutants in the three preceding columns. Some of the generated mutants are not semantically correct and thus did not

IEEE COMPUTER SOCIETY

Table 1. Mutant statistics

Implementation	term.	error	non-term.	total	eq.
Commons.Math	175	339	20	611	27
Jama	102	300	12	468	27
Geostoch	82	223	6	357	24
Flanagan	221	432	25	782	39
Squire	83	225	18	400	24
JScience	10	0	0	10	10
total	673	1519	81	2628	151

compile. Therefore, the total number of generated mutants is larger than the sum of the values in the preceding three columns. Note that only "traditional" mutants were generate by MuJava—class-based ones were omitted since the implementations are not really object-oriented.

Unfortunately, MuJava cannot process Java 5 compatible source code, especially source code using generics. As the JScience library heavily uses generics, the mutants for this implementation had to be created manually.

Often, different mutations lead to equivalent programs. Here, programs which yield the same output for every input are said to be equivalent. As equivalence is undecidable in general, programs which yield the same output for a large number of inputs are declared to be equivalent here. Therefore, a total of 10,000,000 randomly generated matrices were passed to the mutants and the original, and those programs which produced the same output for each input were grouped in an equivalence class. The number of equivalence classes for each implementation are also shown in Table 1 (excluding the original). In the study, only one member of each equivalent class was used, because otherwise, when averaging the results, the mean would be biased and equivalent classes with a higher number of members would be preferred. Note that only mutants not equivalent to the original implementation were taken into consideration.

A possible method to automate the generation of test input data is to use Random Testing [21], i. e. randomly generate the test input data. This approach is unbiased, i. e. test input data are not influenced by the tester and his assumptions. Furthermore, it is really simple. Therefore, a high amount of input data can easily be generated. Random Testing has successfully been used to test SQL database systems [32], the robustness of Windows NT applications [18], and Java Just-In-Time (JIT) compiler [37]. In our study, we used Random Testing because of its simplicity. However, our study and its results do not depend on Random Testing.

The study consists of two series of experiments using randomly generated square matrices as input data. In the first series, the size of the matrices was chosen $n = 2, 4, \ldots, 30$. The elements were generated as independent realizations of a uniformly distributed random variable. To avoid numerical problems, the random variable takes real values in $[-1000, 1000]$. The goal of the first part was to examine how the fault detection capabilities depend on the matrix size. The results showed that it is sufficient to choose matrices of size 6×6 in the second part of the study. In both series, 100,000 random matrices were generated for each mutant. For each of these test inputs, follow-up test cases were generated according to the respective metamorphic relation, and the metamorphic relations were evaluated. Additionally, the original implementation was used as a gold standard oracle. In the second series of simulations, the metamorpic relations are examined in detail using randon matices with fixed size $n = 6$.

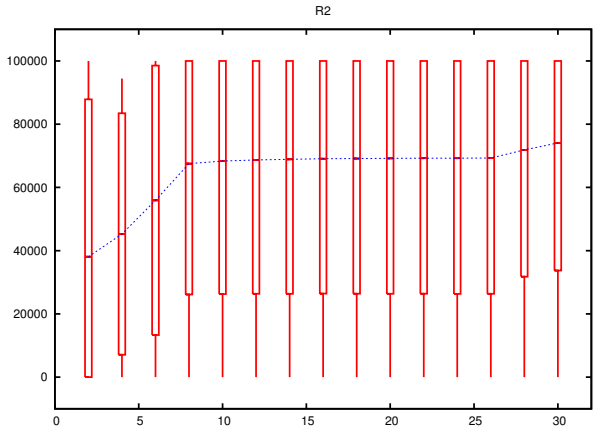

Figure 1. Number of test cases killing a mutant: R2

3.5. Results and Discussion

Figures 1–5 show the results of the first part of the study. The results were taken from the Commons.Math library, the other implementations yield qualitatively the same results. In the figures the matrix size n is shown on the abscissa. On the ordinate, the number of test cases killing a mutant is depicted. The figures present the aggregated results of 27 mutants. For each matrix size, the plot ranges from the minimum number to the maximum number of test cases killing a mutant—connected by a straight line. The mean number of test cases is shown in the middle of a box, these mean values are additionally connected by a dotted line. The box illustrates mean $\pm$ standard deviation. Additionally, the box is limited to the interval $[0, 100000]$.

The first observation is that for $n = 2$, the number of test cases killing a mutant is not satisfying for all relations, but for $n \geq 4$, the results do not change much for relations such as R10, R11 and R12.

The plot for relation R2 (Figure 1) is an example for most

IEEE COMPUTER SOCIETY

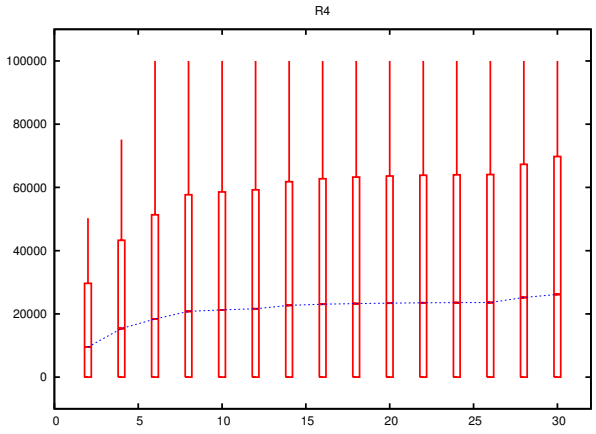

Figure 2. Number of test cases killing a mutant: R4

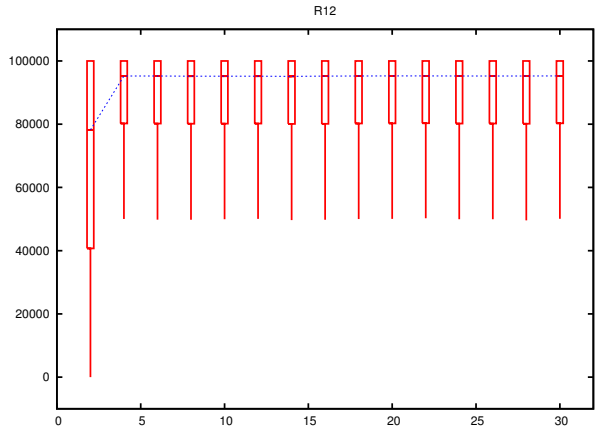

Figure 4. Number of test cases killing a mutant: R12

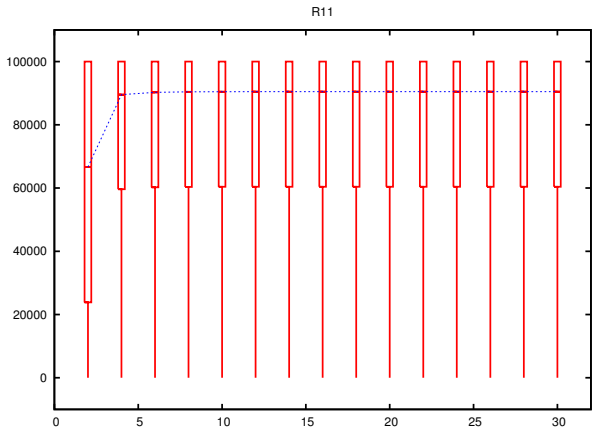

Figure 3. Number of test cases killing a mutant: R11

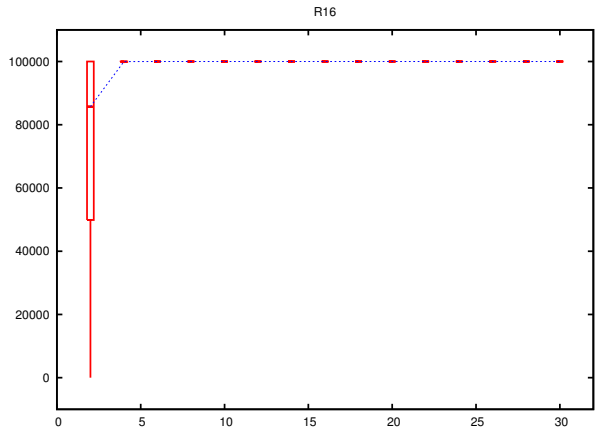

Figure 5. Number of test cases killing a mutant: R16

of the evaluated relations. The mean number of test cases killing a mutant is not constant with respect to the matrix size and in the mean, just about 60% of the test cases killed a mutant. Additionally, the number of successful test cases ranges from 0 to 100,000. The same applies to R1, R3, R5, ..., R9.

For relation R4 (Figure 2), the results are even worse. The average number of test cases killing a mutant is below 25% for each matrix size. This can be explained by the fact that all implementations use Gaussian elimination and relation R4 is used in this algorithm.

Relations R10 and R11 (Figure 3) differ from the previous relations. The mean number of test cases killing a mutant is above 90%. In fact, only one mutant could not be killed, the others were killed by almost 100% of the test cases.

For relation R12 (Figure 4), the minimum number of test cases killing a mutant is almost 50,000. This is nearly equal to the performance of the gold standard oracle. Therefore, R12 performs as good as a gold standard oracle. Note that the mean number of test cases killing a mutant does not depend on the matrix size.

Relation R16 (Figure 5) is an example for a combined relation. For R16, the minimum number of successful test cases is almost 100,000. At least for $n \geq 4$, nearly all test cases kill the mutant. The same applies for R13, R14, and R15. The similar behaviour of R13, R14, and R15 can be explained by the fact that the results of the combinations are mostly influenced by the well performing relations R10, R11, and R12.

Figures 1–5 show that for the most of the relations, the number of test cases killing a mutant is stable for matrix

IEEE COMPUTER SOCIETY

Table 2. Number of test cases killing a mutant: JAMA library

Relation	min	max	mean	std. dev.
R 1	0	100000	74582	36284
R 2	0	98426	42549	42873
R 3	0	98392	79519	29911
R 4	0	100000	26912	37195
R 5	0	100000	71893	33349
R 6	0	100000	61908	32818
R 7	0	100000	76099	34682
R 8	0	100000	83436	30775
R 9	0	100000	74103	37711
R 10	0	100000	92747	25707
R 11	0	100000	93017	25746
R 12	49835	100000	90005	20690
R 13	97198	100000	99704	811
R 14	98568	100000	99841	428
R 15	99824	100000	99984	47
R 16	99942	100000	99995	15
GS	50087	100000	93338	17553

Table 3. Number of test cases killing a mutant: Commons.Math library

Relation	min	max	mean	std. dev.
R 1	0	100000	69369	42533
R 2	0	100000	55910	42613
R 3	0	100000	82124	31947
R 4	0	100000	18415	32941
R 5	0	100000	65105	39499
R 6	0	100000	48083	35761
R 7	0	100000	64060	40705
R 8	0	100000	81619	34045
R 9	0	100000	77748	36145
R 10	0	100000	90051	29970
R 11	0	100000	90245	30012
R 12	49962	100000	95239	15036
R 13	97255	100000	99790	684
R 14	98550	100000	99884	374
R 15	99814	100000	99987	43
R 16	99946	100000	99996	12
GS	49846	100000	95212	15072

Table 4. Number of test cases killing a mutant: JScience library

Relation	min	max	mean	std. dev.
R 1	0	100000	83892	33282
R 2	0	94199	36811	40711
R 3	49753	98327	81609	18183
R 4	0	83332	21638	32394
R 5	0	100000	71789	32179
R 6	0	99996	63263	28248
R 7	0	100000	83401	30878
R 8	33574	100000	88291	24747
R 9	26525	100000	79508	32098
R 10	98357	100000	99835	520
R 11	99013	100000	99901	312
R 12	87542	100000	98754	3940
R 13	99576	100000	99957	134
R 14	99746	100000	99974	80
R 15	99983	100000	99998	5
R 16	99996	100000	99999	1
GS	49948	100000	94994	15828

Table 5. Number of test cases killing a mutant: Matrix class from Michael Thomas Flanagan

Relation	min	max	mean	std. dev.
R 1	0	100000	82477	32700
R 2	0	100000	56470	39883
R 3	0	100000	87032	23370
R 4	0	100000	74773	32566
R 5	0	100000	39169	37859
R 6	0	100000	73811	32056
R 7	0	100000	73081	32739
R 8	1	100000	87467	23420
R 9	0	100000	87933	23801
R 10	0	100000	96291	18543
R 11	0	100000	96373	18546
R 12	49857	100000	94769	15474
R 13	97759	100000	99878	469
R 14	98591	100000	99918	309
R 15	99871	100000	99993	25
R 16	99957	100000	99998	8
GS	49953	100000	96486	12877

size $n \geq 6$. Therefore, in the second part of the study, the matrix size was fixed to $n = 6$. That decreases the runtime of the tests by far, as the time complexity of the Gaussian elimination is $\mathcal{O}(n^3)$ for the given implementations, where n is the matrix size.

In Tables 2–7, the results for each implementation are presented. The minimum number of test cases killing a mutant, the maximum number, the mean, and the standard deviation are provided for each Metamorphic Relation. The input data was also passed to the original implementation—the gold standard—, which is denoted by GS.

For all implementations, the minimum number of test cases killing a mutant is equal to 0 for all relations but for R12. The results for the JScience library seem to differ, but the results are not representative as the mutants were manually created.

The original implementation is sometimes erroneously "killed" by a test case. The reasons for that are numerical problems which can be avoided using integer matrix elements. In this case, the determinant is an integer and can therefore be rounded.

The combination of different relations (R13–R16) does not only improve the minimum number of successful test

IEEE COMPUTER SOCIETY

Table 6. Number of test cases killing a mutant: Matrix class from Jon Squire

Relation	min	max	mean	std. dev.
R 1	0	100000	79617	35205
R 2	0	98409	56399	44262
R 3	0	98405	84104	29271
R 4	0	99987	46333	33037
R 5	0	99925	75522	29025
R 6	0	99999	74036	26499
R 7	0	99995	82596	28230
R 8	0	100000	80042	33588
R 9	0	100000	52202	31738
R 10	0	100000	90757	29383
R 11	0	100000	90819	29398
R 12	50255	100000	97691	10596
R 13	98457	100000	99929	329
R 14	99021	100000	99955	209
R 15	99944	100000	99997	12
R 16	99993	100000	99999	1
GS	50124	100000	96964	11035

Table 7. Number of test cases killing a mutant: GeoStoch library

Relation	min	max	mean	std. dev.
R 1	0	100000	73134	32214
R 2	0	94048	35523	36810
R 3	0	98396	72256	31576
R 4	0	100000	60570	31590
R 5	0	100000	43329	45761
R 6	0	100000	78528	28088
R 7	0	100000	77417	28923
R 8	0	100000	62977	34446
R 9	0	100000	67440	33224
R 10	0	100000	91108	25938
R 11	0	100000	90503	26673
R 12	49878	100000	86523	22129
R 13	93522	100000	99420	1617
R 14	91593	100000	99281	2091
R 15	95124	100000	99687	1217
R 16	95433	100000	99713	1141
GS	49821	100000	89108	20347

cases, but also decreases the standard deviation. Even the additional combination with a bad performing relation as in R16 improves the results.

When using a single relation, R12 seems to be a good choice. The minimum number of test cases killing a mutant is almost equal to that of the gold standard oracle GS, also the mean and the standard deviation are nearly equal to that of GS.

A surprising result is that for combinations of relations the number of test cases killing a mutant is higher than that of a gold standard oracle. This can be explained by the fact that the relations imply a certain number of follow-up test cases. Therefore, more test cases are passed to the mutant than in case of the gold standard oracle. That increases the probability of killing the mutant.

The implicit generation of follow-up test cases has the disadvantage to increase the computational effort. For example R1 implies one follow-up test case for each test case. Relations R10 and R11 actually generate n follow-up test cases, and R12 implies only one follow-up test case for two test cases.

4. General Rules for Assessing Metamorphic Relations

In [11], the evaluation of metamorphic relations is addressed. Two similar case studies based on implementations of shortest resp. critical path algorithms are described. Obviously, if the vertices of a graph are permuted (by whatever permutation), the resulting shortest resp. critical path must have the same length—this is the metamorphic relation used in [11]. Chen et al. therefore define permutations $\pi_1, \ldots, \pi_9$ for their graphs with 10 vertices, where π_i is a cyclic shift by i, yielding 9 metamorphic relations as described. They reason that those relations induced by $\{\pi_1, \pi_3, \pi_7, \pi_9\}$ are the strongest, that of $\{\pi_2, \pi_4, \pi_6, \pi_8\}$ are weaker, and that of $\{\pi_5\}$ is the weakest. E. g. the relations induced by π_1 and π_3 are equivalent, since three applications of π_1 yields π_3 and seven applications of π_3 yield π_1. The surprising result of Chen et al. is that the metamorphic relation induced by π_5 is the best, although it is the mathematically weakest. Therefore, they conclude that blackbox criteria are not suitable and propose to classify metamorphic relations according to their ability to trigger different execution paths, which requires knowledge of the implementation or at least the algorithm.

From the experiments in the present paper, we agree that it is important not to select metamorphic relations too close to the implementation, such as R2, R4, and R8 in our experiments. However, we cannot agree with the conclusion of Chen et al. [11] regarding blackbox criteria. They regard a metamorphic relation MR1 as mathematically stronger than MR2 if MR1 holds for all input-output tuples implies MR2 holds for all input-output tuples, where MR1 and MR2 are implications such as

$$(i_1, \ldots, i_k) \in MR_{j,in} \Rightarrow (f(i_1), \ldots, f(i_k)) \in MR_{j,out},$$

$j = 1, 2$. In the field of software testing, it has no meaning whether a relation MR1 is mathematically stronger than MR2 or not, since we cannot check a property MR1 *for all input-output tuples* with software tests. It could be even possible that a fault which can be detected by MR2 with several input-output tuples can be detected by MR1 with only just one input-output tuple if MR1 is mathematically

IEEE COMPUTER SOCIETY

stronger than MR2. For this reason, it is not useful to classify metamorphic relations according to their "mathematical strength". Only, if there is a "tuple-wise" implication between metamorphic relations, i. e. if one input-output tuple is in MR1, it must also be in MR2, this could be used to classify metamorphic relations. However, such a strong implication will not likely be found. Furthermore, Chen et al. [11] did their experiments with structural similar metamorphic relations, more precisely, with different metamorphic relations induced by different permutations—cyclic shifts. Summarizing, their reasoning about blackbox criteria cannot be upheld.

We used 16 metamorphic relations. Seven relations were structurally completely different, further five relations could be obtained by the column-row analogy, and four relations were combinations of several "basic" relations. From the evaluation of the presented metamorphic relations, the following general rules can be derived, which can be applied to quickly and roughly judge the suitability of metamorphic relations:

1. *Equalities:* Metamorphic relations that have the form of equalities such as R1, R8, and R9 are especially weak. They cannot detect if the result is multiplied by a factor or if a constant is added to the result. Also if the erroneous multiplication or addition takes place in some cases only, these cases cannot be judged correctly by such a metamorphic relation. Symmetric properties, among others, fall into this category.

2. *Equalities of Linear Combinations:* If the metamorphic relation is an equation with linear combinations on each side (with at least two terms on one of the sides), e. g. R2–R7 and R10–R11, it is not vulnerable to erroneous additions—such as the previous category. However, there is a vulnerability against erroneous multiplications.

3. *Semantically Rich Relations:* Typically good metamorphic relations contain much of the semantics of the SUT. Examples are $k \cdot gcd(a, b) = gcd(k \cdot a, k \cdot b)$, $gcd(a, b) \cdot gcd(c, d) = gcd(ac, bd)$, and R12. Whereas R10 and R11 also contain much of the semantics, they are in the form of an equality of linear combinations (see the second criterion) and therefore not so suitable.

4. *Close to Common Implementations/Algorithms:* If metamorphic relations are close to the strategy used by typical implementations/algorithms, such as R2, R4, and R8, their suitability is also quite limited. For example, the changes made in the follow-up test cases are undone by the implementation. Therefore, the computations for the original test case and the follow-up test cases are basically the same. The criterion of similar execution paths by Chen et al. [11] is thus refined.

From our experiments, we can conclude that the above criteria give a rough impression of the performance of a metamorphic relation. Of course, further evaluation e. g. using mutation analysis is necessary to study the relations in more detail. It has also proven beneficial to combine several different good metamorphic relations. Such combinations can avoid the weaknesses of individual metamorphic relations—of course, at the expense of higher costs.

5. Conclusion

Metamorphic Testing is an approach to generate follow-up test cases and to evaluate tuples of "original" and follow-up test cases. It is based on so-called metamorphic relations. In our empirical study, we examined several common Java implementations of determinant computation. We identified a lot of metamorphic relations and classified them using mutation analysis. From our experiments, we could devise a number of criteria to assess metamorphic relations quickly based on the specification. We could show that the reasoning in [11] regarding such blackbox criteria cannot be uphold. Furthermore, we confirmed the usefulness of whitebox considerations proposed in [11] and refined their informal criterion ("different execution path") to criteria underlying or used in the implementation/algorithm. Altogether, we could provide some simple criteria to quickly evaluate metamorphic relations according to their potential usefulness. Furthermore, our study has shown that Metamorphic Testing is a very helpful approach to solve the oracle problem in a lot of cases, since it is often much simpler to give metamorphic relations for a number of inputs (and outputs) than criteria for a single input-output pair.

Acknowledgement

The authors are indebted to T.Y. Chen and Franz Schweiggert for many fruitful discussions.

References

[1] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4), 1988.

[2] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411. ACM, 2005.

[3] Apache Software Foundation. Apache Jakarta Commons.Math. http://jakarta.apache.org/commons/math/.

[4] R. V. Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 1999.

[5] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21st ACM Symposium on the Theory of Computing (STOC 1989)*, pages 86–97, 1989.

IEEE COMPUTER SOCIETY

[6] M. Blum and S. Kannan. Designing program that check their work. *Journal of the ACM*, 41(1):269–291, 1995.

[7] F. Chan, T. Y. Chen, S. C. Cheung, M. Lau, and S. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the 1998 IASTED Conference in Software Engineering*, pages 191–197. Acta Press, 1998.

[8] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, pages 241–249, Los Alamitos, California, 2005. IEEE Computer Society Press.

[9] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[10] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: A case study. In *Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2002)*, pages 327–333. IEEE Computer Society, 2002.

[11] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering*, pages 569–583, Madrid, Spain, 2004. Polytechnic University of Madrid.

[12] T. Y. Chen, T. Tse, and Z. Zhou. Fault based testing in the absence of an oracle. In *Proceedings of the 25th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2001)*, pages 172–178. IEEE Computer Society, 2001.

[13] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: An integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 191–195, New York, 2002. ACM Press.

[14] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM, 2000.

[15] J.-M. Dautelle. JScience. http://jscience.org/.

[16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.

[17] M. T. Flanagan. Java library. http://www.ee.ucl.ac.uk/∼mflanaga/java/Matrix.html.

[18] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 59–68, 2000.

[19] A. Gotlieb. Exploiting symmetries to test programs. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)*, pages 365–374. IEEE Computer Society, 2003.

[20] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2003)*, pages 34–40. IEEE Computer Society, 2003.

[21] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[22] D. Hoffman. Heuristic test oracles. *Software Testing and Quality Engineering*, 1(2):29–32, 1999.

[23] JAMA Project. Java matrix package (JAMA). http://math.nist.gov/javanumerics/jama/.

[24] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[25] J. Mayer. On testing image processing applications with statistical methods. In *Proceedings of Software Engineering 2005, Lecture Notes in Informatics P-64*, pages 69–78. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2005.

[26] J. Mayer. Towards a reliable statistical oracle and its applications. *Softwaretechnik-Trends*, 25(1):21–27, 2005.

[27] J. Mayer and R. Guderlei. Test oracles using statistical methods. In *Lecture Notes in Informatics P-58*, pages 179–189. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2004.

[28] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2000.

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

[30] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. An experimental mutation system for Java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[31] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55, 2000.

[32] D. Slutz. Massive stochastic testing of SQL. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB 1998)*, pages 618–622, 1998.

[33] J. Squire. A Java matrix class. http://www.csee.umbc.edu/∼squire/download/Matrix.java.

[34] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 458–466, Los Alamitos, California, 2004. IEEE Computer Society Press.

[35] Ulm University, Dept. of Stochastics and Dept. of Applied Information Processing. GeoStoch homepage. http://www.geostoch.de.

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

[37] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)*, pages 20–24. IEEE Computer Society, 2003.

[38] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, Japan, 2004. Software Engineers Association.

IEEE COMPUTER SOCIETY