To appear in *Proceedings of the 8th International Symposium on Future Software Technology* (*ISFST 2004*),
Software Engineers Association, Japan (2004)

Metamorphic Testing and Its Applications *†

Zhi Quan Zhou ‡‖, D. H. Huang ‡, T. H. Tse §,
Zongyuan Yang ¶, Haitao Huang ¶, and T. Y. Chen ‡

‡ *School of Information Technology*
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
Email: {zhzhou, dhuang, tchen}@it.swin.edu.au

§ *Department of Computer Science*
The University of Hong Kong
Pokfulam Road, Hong Kong
Email: thtse@hku.hk

¶ *Department of Computer Science*
East China Normal University
3663 Zhongshan(N) Road
Shanghai 200062 P.R. China
Email: {yzyuan, hthuang}@cs.ecnu.edu.cn

ABSTRACT

An "oracle" in software testing is a procedure by which testers can decide whether the output of the program under testing is correct. In some situations, however, the oracle is not available or too difficult to apply. This is known as the "oracle problem". In other situations, the oracle is often the human tester who checks the testing result manually. The manual prediction and verification of program output greatly decreases the efficiency and increases the cost of testing.

A metamorphic testing method has been proposed to test programs without the involvement of an oracle. It employs properties of the target function, known as metamorphic relations, to generate follow-up test cases and verify the outputs automatically. In this article, we shall present the concepts, procedures, and applications of metamorphic testing.

Keywords

Metamorphic testing, metamorphic relation, oracle, successful test case, automated testing, multiple executions.

INTRODUCTION

The verification of program correctness plays a critical role in software development. In the past decades, it is shown that the use of formal verification, i.e., program proving, to real-life applications has been very limited [12] owing to the difficulties of proofs and automation. Software testing [2], therefore, remains the most popular means for practitioners to check the correctness of their programs, although testing cannot prove the absence of errors in most situations [2, 14].

There is an *oracle assumption* in the theory of testing [18]. Let $p(x)$ be a program implementing function $f(x)$ on domain D. To test this program, the tester runs p on T, a set of *test cases*: $T = \{t_1, t_2, \ldots, t_n\} \subset D$, where $n \geq 1$. The outputs $p(t_1), p(t_2), \ldots, p(t_n)$ are then checked against the expected results $f(t_1), f(t_2), \ldots, f(t_n)$, respectively. If $p(t_i) = f(t_i)$, then t_i is called a *successful* test case; if $p(t_i) \neq f(t_i)$, then t_i is called a *failure-causing* test case. The mechanism by which the tester can decide whether $p(t_i) = f(t_i)$ for $i = 1, 2, \ldots, n$ is known as the *oracle*. In software testing literature, it is usually assumed that the oracle is available

† This research is supported in part by a discovery grant of the Australian Research Council (Project No. DP 0345147), a grant of the Research Grants Council of Hong Kong (Project No. HKU 7145/04E), and a grant of the University of Hong Kong.

‖ Corresponding author.

and, hence, the mainstream of the researches has concentrated on the development of *test case selection strategies*, that is, the approaches for selecting t_i's that have a higher chance of causing a failure.

In some situations, however, the oracle is not available or too expensive to apply. This is known as the *oracle problem* [18]. For example, the outputs of programs conducting complicated computations, such as numerical integrations, are difficult to verify; In multiple precision arithmetic, the operands involved are very large numbers and, hence, the results are very expensive to check; When testing a compiler, it is not easy to verify whether the generated object code is equivalent to the source code; When testing object-oriented programs, it is very difficult to decide whether two objects are equivalent. Other examples include testing programs performing simulations, conducting combinatorial calculations, drawing graphs to the monitor, etc. On the other hand, when the oracle is available, if it is a human tester, the manual predictions and comparisons of the test results are often time consuming and error prone [13, 15]. As a matter of fact, the oracle problem has been "one of the most difficult tasks in software testing" [15].

Another important topic is how to effectively utilize the successful test cases. This is because, even when the oracle is available, testing is still very expensive and takes a major part in the total development cost [12] because test case design and implementation are always labor intensive. Hence, it is important to make the best of each test case. Having said that, it must be pointed out that the majority of the test cases are successful test cases that do not reveal any failure. In conventional testing, these test cases are considered useless and, hence, discarded [17] or merely kept for regression testing later. The theory of fault-based testing [16] is a breakthrough because it employs successful test cases to prove the absence of certain types of faults. Unfortunately, most testing techniques are not fault-based and most test cases do not reveal any failure. As a result, useful information carried in those successful test cases remains unexploited.

A *metamorphic testing method* (MT) has been proposed by Chen et al. [5] to employ successful test cases and alleviate the oracle problem. Based on the successful test cases, follow-up test cases can be generated by making reference to *metamorphic relations*, that is, relations among *multiple* executions of the target program. The generation of the follow-up test cases and verification of the test results do not require an oracle. In this article, we present the basic concepts of metamorphic testing and introduce a range of its applications.

BASIC CONCEPTS OF METAMORPHIC TESTING

Metamorphic testing (MT) [5] is a technique to generate *follow-up* test cases based on existing test cases that have not revealed any failure. MT should be applied in conjunction with other test case selection strategies that generate the initial set of test cases. Let us consider a program p implementing function f on domain D. Let S be the test case selection strategy adopted by the tester, such as data flow testing or branch coverage. According to S, a test set $T = \{t_1, t_2, \ldots, t_n\} \subset D$, where $n \geq 1$, can be generated. Running the program on T yields the outputs $p(t_1), p(t_2), \ldots, p(t_n)$. When there is an oracle, these test results can be verified against $f(t_1), f(t_2), \ldots, f(t_n)$; otherwise the tester may still have some way to identify some outcomes that are obviously wrong. For example, an execution that runs too long can be considered a failure; when a trigonometric function computing $\cos x$ returns a value greater than 1, a failure can also be found immediately [18]. When a failure has been detected, testing can stop and the program will be debugged; otherwise T is a set of successful test cases. In the latter case, MT can be applied to automatically generate follow-up test cases $T' = \{t'_1, t'_2, \ldots, t'_n\} \subset D$ based on the initial successful test set T, so that the program can be further verified against some necessary properties. MT is useful because the vast majority of test cases are successful ones — although they have not revealed any failure, these test cases do carry useful information ignored in conventional testing.

MT generates follow-up test cases by making reference to "metamorphic relations" (MR). For program p, an MR is a property of its target function f. The unique character of MR is that it involves *multiple* executions. For example, if $f(x) = e^x$, then the property $e^a \times e^{-a} = 1$ is a typical MR. Hence, for a successful test case, say $t_i = 0.3$, metamorphic testing generates its follow-up test case $t'_i = -0.3$ and then runs the program again on t'_i. Finally, the relation of the two outputs are checked against the expected relation $p(0.3) * p(-0.3) = 1$. [1] If this identity does not hold, then a failure is immediately detected. Like all the other testing methods, however, the conditions checked by MT are necessary, but may not be sufficient for program correctness.

Because MT checks the relations among several executions rather than the correctness of individual outputs, MT does not need an oracle and can be fully automated. This methodology has also been applied to fault-based testing without oracles [8]. Further study has also been conducted in [11], where an experimental MT framework is constructed.

In fact, identity relations like $e^x \times e^{-x} = 1$ have long been used in practice to test programs, especially in the area of numerical computing (such as [9]). Apart from conventional testing, identity relations have also been used for fault tolerance in run time [1]. The techniques of *program checker* [3] and *self-testing/correcting* [4] also intensively involve the use of identity relations of the target function. Nevertheless, there are notable differences between these methods and metamorphic testing. Firstly, MT can be used in conjunction with other test case selection strategies, including both black-

[1] For floating point computation, some rounding error will be allowed.

and white-box testing strategies. When the initial test set has not revealed any failure, MT can be used to further exploit the useful information carried in the successful test cases to generate follow-up accompanying test sets so that the program can be verified further against necessary properties efficiently and automatically. Secondly, metamorphic relations are not limited to identity relations. It includes inequalities, subsumption relations, and convergence properties to name a few.

APPLICATION OF MT TO NUMERICAL PROBLEMS

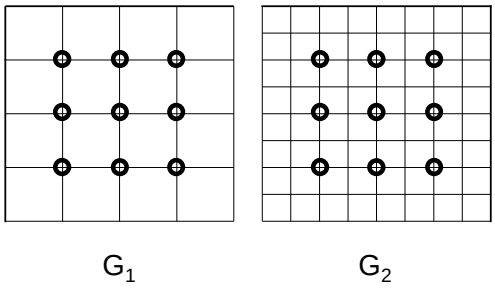

Figure 1: Metamorphic test cases

When testing numerical programs whose outputs are not easy to verify, a frequently adopted approach is to use special or simple values as inputs [18]. For example, when testing a program computing the sine function, special values such as 0, $\pi/4$, $\pi/2$, etc., are always standard test cases. These special or simple inputs, however, are not enough in building people's confidence in the correctness of their programs on more complex and random inputs. We have done experiment in [7], and found cases where errors could not be detected by special values. When special-value testing is combined with metamorphic testing, on the other hand, more subtle faults can be revealed [7]. For example, a program $p(x)$ for sine function could compute correctly on a test case x_0. However, with a follow-up test case generated based on the MR $\sin(x+\pi) = -\sin(x)$, a failure will be immediately detected when $p(x_0+\pi) \neq -p(x_0)$.

As discussed earlier, metamorphic relations are not limited to identity relations. We would like to cite one of our examples in [6] to illustrate how to use the convergence property to test programs that solve partial differential equations. We use a program adapted from [10] that solves the following thermodynamic problem: For an insulated plate in rectangular shape with homogeneous boundary temperatures along each edge, we want to know the temperature of each point on the plate after the heat potential has reached stability.

To solve this problem, the program uses the "alternating direction implicit" method to solve the Laplace equation with Dirichlet boundary conditions. We created a mutant of the original program by replacing the correct statement

```
if ( fabs ( uMat[i][j] - vMat[j][i] > larg )
    larg = fabs ( uMat[i][j] - vMat[j][i] );
```

with

```
if ( fabs ( uMat[i][j] - uMat[j][i] > larg )
    larg = fabs ( uMat[i][j] - vMat[j][i] );
```

The above fault is quite subtle, and there is no oracle to test this program. The mutant program gives identical outputs as the correct program when running on 3×3 and 7×7 mesh grids. Both programs also return very close results on 15×15 mesh grids. In addition, we have also tested the program on the following special inputs: (1) All the four edges have an equal temperature; (2) Assign equal length to all the edges and use symmetric boundary conditions. It is therefore expected that the distribution of the temperatures should be symmetric as well; (3) Use symmetric boundary conditions wrt both the *x*- and *y*-axes, respectively. The test result is that none of the above special values could detect a failure.

Now let us verify the program using metamorphic testing method. The convergence property of the solutions can be identified as a metamorphic relation [6]. For any given point p, let $T_{G_i}(p)$ denote its temperature calculated by the program using a mesh grid G_i. If we use G_i, G_j, and G_k to denote any mesh grids, then the following metamorphic relation can be identified [6]:

$$G_i \subset G_j \subset G_k \rightarrow$$
$$T_{G_i}(p) \leq \min\{T_{G_j}(p), T_{G_k}(p)\} \text{ or}$$
$$T_{G_i}(p) \geq \max\{T_{G_j}(p), T_{G_k}(p)\}.$$

The program is then tested against this MR. The temperatures of the same 9 points $p_1, p_2, \ldots, p_9$ have been observed using mesh grids $G_1, G_2, \ldots, G_5$, where $G_1 \subset G_2 \subset \ldots \subset G_5$. For example, Figure 1 shows G_1, G_2, and the 9 points. When we check the differences between the computed results against the MR, a failure can be detected easily as the expected inequality is violated.

APPLICATION OF MT TO NON-NUMERICAL PROBLEMS

Metamorphic testing is not limited to numerical programs only. In fact, metamorphic relations can be identified in almost every area. In this section, we shall give some but a few examples to illustrate how to employ MT in non-numerical areas.

Graph Theory

A lot of graph theory problems are combinatorial problems. As a result, it is very expensive to verify the outputs when the input graph is nontrivial.

Let us take the Shortest Path problem in an undirected graph as an example. When the test case is a nontrivial graph like that shown in Figure 2, there is no oracle efficiently

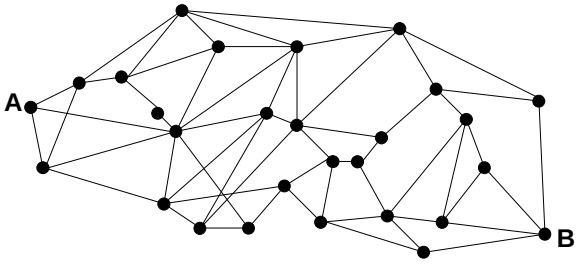

Figure 2: A nontrivial graph (Weights of edges are omitted)

applicable to verify whether the returned set S_{AB}, all shortest paths between A and B found by the program, are indeed the shortest ones or whether this set is complete, where $S_{AB} = \{P_1, P_2, \ldots, P_n\}$ and $n \geq 1$.

In this situation, MT can help by checking the program against selected MRs. A popular property that can be identified for graph theory problems is permutation. If G is the first test case, then let G' be a permutation of G. Running the program again on G' should produce the same output as produced on G.

Another MR can be identified as follows: Randomly select an element P_i from S_{AB}. Hence, P_i is supposed to be one of the shortest paths from A to B. Randomly select a vertex X in path P_i. Then, run the program to get the shortest paths from A to X and from X to B, respectively. Suppose the outputs are S_{AX} and S_{XB}, respectively. One of the expected MRs is that, for any path $Q \in S_{AX}$ and any path $Q' \in S_{XB}$, the concatenation of Q and Q' must be an element in S_{AB}.

Computer Graphics

When the outputs of a program involve a large amount of data, they are expensive to verify. For example, computer graphics software generates graphics and prints them on the screen. It is, however, practically impossible for the tester to manually check whether each and every pixel is displayed properly. In this situation, a practical approach is that after checking the correctness of certain amount of individual outputs, we apply MT to verify all the outputs in a more cost-effective way as follows.

Figure 3 illustrates a graph generated by a realistic-graphics-generation software. Note that this figure is simplified for illustration purpose only. For the tester, it is not easy to verify whether all the pixels in the screen are displayed properly because the generation of realistic graphics involves complicated computation and there is a huge amount of pixels. Nevertheless, some metamorphic relations can be identified. For example, if the position of the light source changes, then the brightness of all the points that become closer to the light source will increase according to a certain formula; similarly,

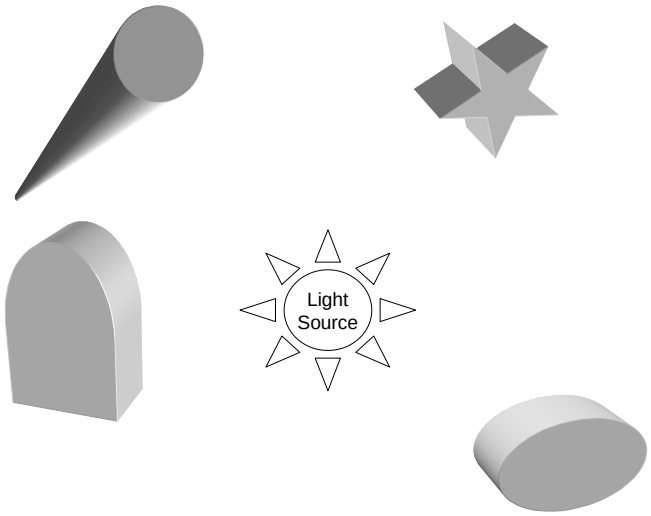

Figure 3: Computer graphics

all the points that become farther will become darker. This is an easy approach to check all the displayed pixels quickly and automatically. Following this way, many other metamorphic relations can be identified as well.

Compilers

Testing compilers is tough. This is because the equivalence between the source code and the object code is difficult to verify. In this subsection, we give an example to illustrate how to use MT to alleviate this problem.

Suppose *cp* is a parallelizing compiler. Suppose we have the following source code as a test case:

```
     int a, b, c, d;
1    read(a, b);
2    c = a + 1;
3    d = 100;
4    d = d * b;
5    ...
```

Even if we do not know whether the output object code is correct, we can still identify metamorphic relations to test the compiler. As a simple example, we can find that statement 2 and statements (3, 4) are independent of each other. Hence, we can exchange their sequence to construct a follow-up test case:

```
     int a, b, c, d;
1    read(a, b);
2    d = 100;
3    d = d * b;
4    c = a + 1;
```

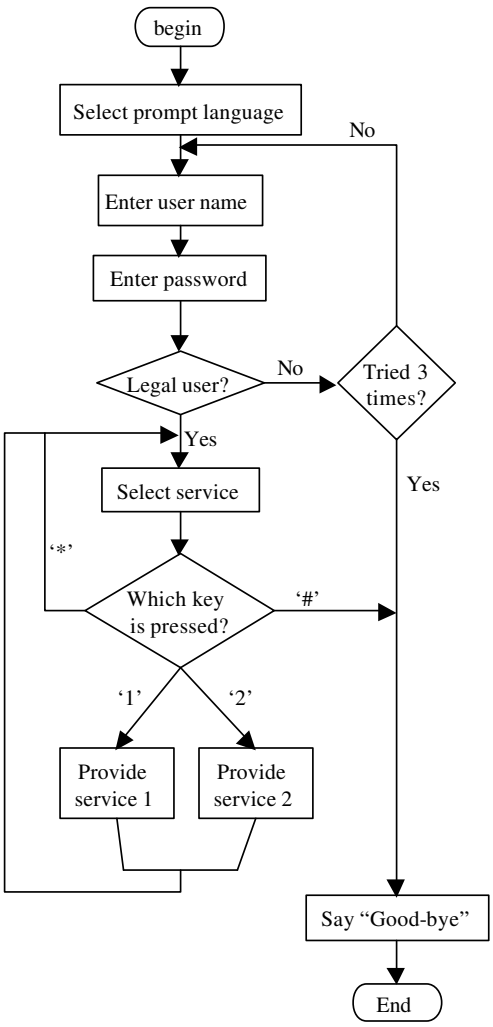

Figure 4: A flowchart of automated telephone service

5 ...

For the above source code, the parallelizing compiler *cp* should detect identical parallelism as for the first one, and this can be verified much more easily than the correctness of the object code.

Interactive Software

For interactive software, the program inputs can be a serial of user actions rather than static data. For example, when testing an Internet browser, the test cases can be HTML files and consecutive user actions as follows: Enter URL → Click "Item 1" → Click "Refresh" → Click "Back" → Select menu "File" → Select menu "Print" → Click "OK" ...

Metamorphic relations can be identified when testing interactive software. In this situation, an MR is a relation among different sequences of user actions and their corresponding outputs. For example, Figure 4 shows an illustrative flowchart for telephone transaction software. When users have dialed in, they will need to select their preferred service language first. Then they will enter their user name and password. In case of a failure, they will have two more chances to try. For legal users, they will be asked to select services: press "1" for Service 1; press "2" for Service 2; press "*" to repeat the voice message, and press "#" to quit. For this kind of software, many different combinations of user actions are expected to produce the same results. For instance, a failed followed by a successful login should be treated the same as a successful login without a failure; doing something and then cancelling it should be treated the same as quitting the program in the beginning; performing Service 2 followed by Service 1 in one dial-in should produce the same result as performing the two transactions separately in two different dial-ins in the same sequence; no matter how many times the users press "*" when selecting services, the final results should be the same ... All these properties can be used as metamorphic relations to test the program automatically.

CONCLUSION

This article has introduced the concepts and a wide range of applications of metamorphic testing. The unique character of MT is that it does not require human involvement to generate follow-up test cases and verify the test results and, hence, it can be fully automated. Because metamorphic relations widely exist in both numerical and non-numerical areas, MT is a practical approach applicable to the vast majority of real-life applications. Also because this method can be combined with any test case selection strategy, MT is a useful approach for practitioners to further exploit their successful test cases. As MRs are identified with regard to the specification, good knowledge of the problem domain is necessary for an effective application of MT.

It should be noted that, because MT checks necessary rather than sufficient properties, and also because it does not check the correctness of individual outputs, pure MT is not enough to establish confidence in the program's correctness with regard to the original specification. Hence, MT should be combined with other testing methods such as special-value testing to achieve the best results. In our future research, we shall investigate how to select the most effective metamorphic relations when there is more than one candidate.

REFERENCES

1. Ammann, P. E. and Knight, J. C. Data diversity: an approach to software fault tolerance, *IEEE Transactions on Computers* 37 (4), 1988, pp. 418–425.
2. Beizer, B. *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990.
3. Blum, M. and Kannan, S. Designing programs that check their work, In *Proceedings of the 31st Annual ACM*

Symposium on Theory of Computing (*STOC '89*), ACM Press, New York, 1989, pp. 86–97. Also *Journal of the ACM*, 42 (1), 1995, pp. 269–291.

4. Blum, M. Luby, M. and Rubinfeld, R. Self-testing / correcting with applications to numerical problems, *Journal of Computer and System Sciences*, 47 (3), 1993, pp. 549–595.
5. Chen, T. Y., Cheung, S. C., and Yiu, S. M. 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, Hong Kong, 1998.
6. Chen, T. Y., Feng, J., and Tse, T. H. Metamorphic testing of programs on partial differential equations: a case study, In *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), IEEE Computer Society Press, Los Alamitos, California, 2002, pp. 327–333.
7. Chen, T. Y., Kuo, F.-C., Liu, Y., and Tang, A. Metamorphic testing and testing with special values, In *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel / Distributed Computing* (*SNPD 2004*), International Association for Computer and Information Science, Mt. Pleasant, Michigan, 2004.
8. Chen, T. Y., Tse, T. H., and Zhou, Z. Q., Fault-based testing without the need of oracles, *Information and Software Technology*, 45 (1), 2003, pp. 1–9.
9. Cody, W. J., Jr. and Waite, W. *Software Manual for the Elementary Functions*, Prentice Hall, Englewood Cliffs, New Jersey, 1980.
10. Gerald, C. F. and Wheatley, P. O. *Applied Numerical Analysis*, Addison Wesley, Reading, Massachusetts, 1999.
11. Gotlieb, A. and Botella, B. Automated metamorphic testing, In *Proceedings of the 27th Annual International Computer Software and Applications Conference* (*COMPSAC 2003*), IEEE Computer Society Press, Los Alamitos, California, 2003, pp. 34–40.
12. Hailpern, B. and Santhanam, P. Software debugging, testing, and verification, *IBM Systems Journal* 41 (1), 2002, pp. 4–12.
13. Hamlet, D. Predicting dependability by testing, In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 1996*), ACM Press, New York, 1996, pp. 84–91.
14. Howden, W. E. Reliability of the path analysis testing strategy, *IEEE Transactions on Software Engineering* **SE-2**, 3, 1976, pp. 208–215.
15. Manolache, L. I. and Kourie, D. G. Software testing using model programs, *Software: Practice and Experience*, 31 (13), 2001, pp. 1211–1236.
16. Morell, L. J. A theory of fault-based testing, *IEEE Transactions on Software Engineering*, 16 (8), 1990, pp. 844–857.
17. Myers, G. J. *The Art of Software Testing*, Wiley, New York, 1979.
18. Weyuker, E. J. On testing non-testable programs, *The Computer Journal*, 25 (4), 1982, pp. 465–470.