Metamorphic Testing and Beyond *

T. Y. Chen †, F.-C. Kuo †, T. H. Tse ‡ §, Zhi Quan Zhou †

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

‡ *Department of Computer Science and Information Systems*
The University of Hong Kong
Pokfulam Road, Hong Kong
Email: tse@csis.hku.hk

Abstract

When testing a program, correctly executed test cases are seldom explored further, even though they may carry useful information. Metamorphic testing proposes to generate follow-up test cases to check important properties of the target function. It does not need a human oracle for output prediction and comparison. In this paper, we highlight the basic concepts of metamorphic testing and some interesting extensions in the areas of program testing, proving, and debugging. Future research directions are also proposed.

Keywords: *Follow-up test cases, metamorphic testing, semi-proving, successful test case, test case selection strategy, testing oracle*

1. Introduction

It is impractical, if not impossible, to test a program with all conceivable inputs [1]. Instead, we should aim at selecting test cases with higher probabilities of revealing program failures. Hence, a lot of research has been done on developing *test case selection strategies.*

A *successful test case* is one on which the program computes correctly. Since successful test cases do not reveal any failure, they are conventionally considered useless [22] and thus discarded by testers or merely retained for reuse in regression testing later. We note, however, that successful test cases *do* carry useful information, albeit seldom explored. Fault-based testing [21], for example, is a significant attempt to make use of such information. In fault-based testing, if a program has successfully passed all the test cases, then it can be guaranteed to be free from certain types of faults. Unfortunately, most testing methods are not fault-based, and most test cases are executed successfully. Thus, some valuable information that results from program testing will remain buried and unused.

Another limitation of software testing is the *oracle problem* [23]. An *oracle* is a mechanism against which people can decide whether the outcome of the program on test cases is correct. In some situations, the oracle is not available or is too expensive to be applied [23]. In cryptography systems, for example, large number arithmetic is usually involved. It is very expensive to verify the correctness of a computed result. Other examples include deciding the equivalence between the source and object codes when testing a compiler; and deciding the correctness of an output when testing a program that performs numerical integration. Furthermore, even when manual prediction and comparison of testing results are possible, they are often time consuming and error prone [18, 20]. The oracle problem is "one of the most difficult tasks in software testing" [20] but is often ignored in the testing theory [18].

A *metamorphic testing* (MT) method has been proposed [4] with a view to making use of the valuable information in successful test cases. It does not depend on the availability of an oracle. It proposes to generate follow-up test cases based on metamorphic relations, or properties among inputs and outputs of the target function. In this

'This research is supported in part by a discovery grant of the Australian Research Council (Project No. DP0345147), a grant of the Research Grants Council of Hong Kong, and a grant of the University of Hong Kong.

§ Contact author.

paper, we would like to highlight the method and explore future research directions.

2. Metamorphic testing

Metamorphic testing is used in conjunction with other test case selection strategies [4]. Given a test case selection strategy S, such as path coverage, a set of test cases $T = \{t_1, t_2, \ldots, t_n\}$, where $n \geq 1$, is generated. The program is then tested on T. If no failure is revealed after running all t_i in T for $i = 1, 2, \ldots, n$, then T will be a set of successful test cases.

At this stage, metamorphic testing can be carried out to generate follow-up test cases according to metamorphic relations. A ***metamorphic relation*** (MR) is an expected relation among the inputs and outputs of ***multiple*** executions of the target program. For a successful test case t_i and a chosen MR, we can construct follow-up test case(s), say t_i', and run the program again. Let p denote the program under test. We check t_i, $p(t_i)$, t_i', and $p(t_i')$ against the MR. If MR cannot be satisfied, the program must have failed.

Consider, for instance, a program that computes the sine function. The property $\sin x = \sin(180 - x)$ can be used as a metamorphic relation. Let $t = 57.3$ be one of the test cases chosen according to a selection strategy such as branch coverage. Suppose the output **is** 0.8415. This output may not be verified easily if an oracle is not available. On the other hand, regardless of whether an oracle exists, MT suggests testing the program with a follow-up test case $180 - 57.3$. The program is run on this test case to produce a second output, say 0.8402. The two outputs are then compared. Obviously, they do not satisfy the expected MR and hence a failure is detected.

It should be noted that the idea of verifying programs against selected properties is not new. It has long been used in practice (see [12], for example). The techniques of ***program checkers*** [2] and ***self-testing/correcting*** [3] also make use of properties that involve multiple executions of the program. There are, however, significant differences between MT and other property-based testing methods. First, before MT is applied, a test case selection strategy S and a set of test cases T corresponding to S must exist in the first place. If no failure is revealed by T, then MT can be applied to generate a new set of test cases as a partner accompanying T, so that the program can be further verified against some necessary metamorphic relations. This is regardless of whether an oracle is available. Another characteristic of MT is that MRs are not limited to identity relations. Any expected relation involving inputs and outputs of two or more executions of the program can be taken as an MR. In [5], for instance, we used the convergence property as a metamorphic relation to test a program that solves a partial differential equation. For more detailed discussions on the differences between MT and other methods, readers may refer to [8].

As has been shown, MT does not check the correctness of individual outputs. Instead, it checks the relations among several executions. Since no manual output predictions and comparisons are required, MT can be efficient and fully automated. In [16], an experimental metamorphic testing framework has been developed to follow up on our study.

Note that, since an MR is a necessary property, it may not be sufficient for program correctness. This is indeed a limitation of all testing methods.

3. Interesting results and potential research directions

3.1. Testing in the absence of an oracle

In general, when oracles are not available, testers often test the programs using special or simple values for which correct results are actually known [23]. Our experimental results in [7] showed, however, that these special and simple values are not enough in revealing program defects. By incorporating MT, the problem can be better tackled.

In [7], we studied a faulty program purportedly computing the sine function. We first tested this program using the following 5 special values, for which the sine function values are well known: 0, $\pi/6$, $\pi/4$, $\pi/3$, and $\pi/2$. The outputs computed by the program, however, are all correct.

We then identified 10 metamorphic relations to generate metamorphic test cases [7]. These MRs are:

R_1 : $\sin x - \sin(x + 2\pi) = 0$;

R_2 : $\sin x + \sin(x + n) = 0$;

R_3 : $-\sin(-x) - \sin x = 0$;

R_4 : $\sin x - \sin(\pi - x) = 0$;

R_5 : $\sin x + \sin(2\pi - x) = 0$;

R_6 : $\sin x + \sin y + \sin z - \sin(x + y + z) - 4\sin((x+y)/2)\sin((x+z)/2)\sin((y+z)/2) = 0$

R_7 : $\sin^2 x + \sin^2(\pi/2 - x) - 1 = 0$;

R_8 : $\sin 3x - 3\sin x + 4\sin^3 x = 0$;

R_9 : $\sin^2 x - \sin^2 y - \sin(x + y)\sin(x - y) = 0$;

R_{10} : $\sin 5x - 16\sin 5x - 5\sin 3x + 10\sin x = 0$;

For each of the 5 special values, follow-up test cases were generated to verify the program against the 10 MRs R_1, R_2, ..., R_{10}, respectively. After taking into consideration rounding errors in floating-point computation, the results were as follows: For the special value "0", two MRs were violated; for "$\pi/6$", four were

violated; for "$\pi/4$", six were violated; for "$\pi/3$", six were violated; and for "$\pi/2$", seven were violated. This result shows that we should not stop when a program has been tested on some special values and no failure has been revealed. By making reference to metamorphic relations, follow-up test cases can be constructed and the program tested further. This will increase the chance of revealing defects in the program. We also see from the results that, for a given metamorphic relation, some inputs may cause a failure while others may not. When performing MT, therefore, the test cases should include both special values (for which an oracle exists) and random values (for which an oracle does not exist), in order to maximize the possibility of revealing a failure.

Our results in [7] also show that the failure-causing abilities of different MRs vary greatly. After metamorphic testing based on the 5 special values, the failure rates of R_1, R_2, ..., R_{10} are 0, 0.8, 0.6, 0.4, 0.4, 1, 0, 0.6, 0.4, and 0.8, respectively. This result demonstrates that, when performing MT, we should try to employ more than one MR because different MRs may have varying failure-causing abilities for different types of defect.

3.2. Beyond identity relations

In this section, we shall continue our discussions on testing in the absence of an oracle.

Metamorphic relations are not limited to identity relations. In [5], for instance, we used the convergence property as an MR to test a program solving a partial differential equation. The program was adapted from [15]. It attempts to solve the following thermodynamic problem: Suppose we are given an insulated rectangular plate. Its boundary temperatures are homogeneous along each edge. After the heat potential of the plate has reached stability, we would like to find the temperature of each point on the plate.

The program calculates the temperatures on the plate by solving a Laplace equation with Dirichlet boundary conditions. The algorithm uses the "alternating direction implicit" method. We have seeded a fault into the 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] );
```

It is difficult to verify the results of computation because of the lack of a testing oracle. We used both simple and special values as test cases for the faulty program but no failure was revealed. For example, it produces exactly the same outputs as the correct program when computing on

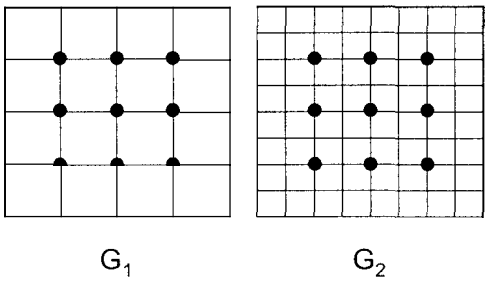

Figure 1. Examples of metamorphic test cases

3 x 3 and 7 x 7 mesh grids. It produces a result fairly close to the one computed by the correct program when a 15 x 15 mesh grid is used. We also used the following special cases to test the program: *(i)* setting the temperatures at all edges to be equal; *(ii)* using a square plate and setting the boundary conditions to be symmetric, hence producing a symmetric distribution of the temperatures; *(iii)* setting the boundary condition to be symmetric with respect to both the horizontal and vertical axes. **All** these special cases cannot reveal the fault in the program.

We tested this program using the convergence property of the solutions of partial differential equations [5]. Let us use $T_{G_i}(P)$ to denote the temperature at a point P computed by the program using a mesh grid G_i. Let G_i, G_j, and G_k denote any three mesh grids. We identified and proved the following MR for testing the program:

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

Using this convergence property as the MR, we tested the program at the same 9 points P_1, P_2, ..., P_9 using mesh grids G_1, G_2, ..., G_5, where $G_1 \subset G_2 \subset \ldots \subset G_5$. Figure 1, for example, shows the 9 points for mesh grids G_1 and G_2. By comparing the differences between the 5 computation results, it can be found that this series of outputs do not satisfy the expected MR. Hence, a failure has been revealed.

3.3. Fault-based testing without oracles

We have also applied the technique of metamorphic testing to fault-based testing [9] so that prescribed faults can be eliminated from the program even when an oracle is not available.

The theory of fault-based testing was introduced by Morell [21]. He observed that, although testing could not

```
    double Power (double u, double v) {
    double uMinusOne, numerator, lnTerm, result;
    int i;
1.  if (v == 0)
2.      result = 1;
    else {
3.      if ((int)v == v) && (v>0)) {
4.          result = 1;
5.          for (i=1; i<=v; i++)
6.              result = result * u;
        }
        else {
        /* ln(u) = ln(1 + (u-1)) = (u-1) - 1/2 * (u-1)^2 +
        1/3 * (u-1)^3 - ... */
7.          i = 1;
8.          uMinusOne = u - 1;
9.          numerator = uMinusOne;
10.         lnTerm = uMinusOne;
11.         result = uMinusOne;
12.         while (AbsoluteValue (lnTerm) > 1e-16) {
            /* 1e-16 = 10^-16 */
13.             i = i + 1;
14.             numerator = (-1) * numerator * uMinusOne;
15.             lnTerm = numerator / i;
16.             result = result + lnTerm;
            }
17.         result = exp (v * result);
        }
    }
18. return result;
}
```

Figure 2. Program *Power*

prove the correctness of programs, correctly executed test cases can indeed show that the program code is free from certain types of fault. Like other testing methodologies, Morell's method also requires an oracle. This is because we must know whether the test cases have been executed correctly in the first place.

By employing metamorphic relations, the oracle problem in fault-based testing can be alleviated [9]. Figure **2** shows a program that has been used as one of the examples to illustrate our method. For the given input u and v, the program computes u^v. If $v = 0$, it will immediately return "1". If v is a positive integer, then it produces the result by multiplying u by itself v times. Otherwise, it uses the formula $u^v = e^{v\ln(u)}$ to compute the power.

Suppose we would like to know whether statement 11 is correct with respect to a constant substitution for "uMinusOne". In other words, we would like to know whether the correct statement 11 could have been replaced erroneously by

```
result = F;
```

where "F" is a constant value instead of the variable "uMinusOne". To achieve this goal, the fault-based testing technique should be used. Because of a lack of an oracle

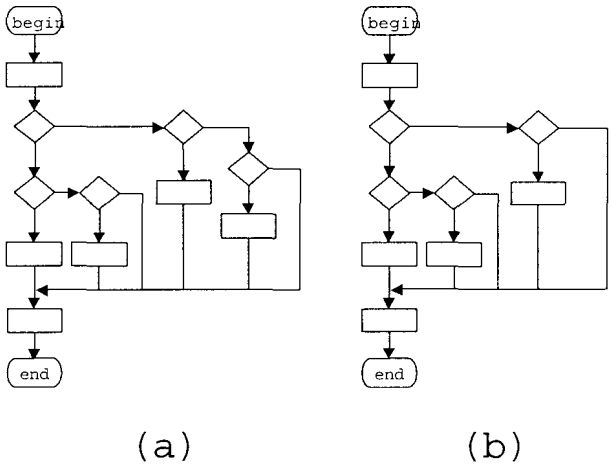

Figure 3. Control flow of program *Med*

for this problem for the general situation, however, Morell's method could not be applied. On the other hand, by making use of the simple metamorphic relation "$u^v \times u^v = (u \times u)^v$", the problem can be solved. It can be proved that a constant substitution for "uMinusOne" is impossible without causing a failure [9]. We have also given more examples to demonstrate the use of both actual and symbolic test cases in eliminating prescribed faults.

3.4. Metamorphic testing using symbolic inputs

Metamorphic testing is not restricted only to actual inputs. With the use of *symbolic inputs,* metamorphic testing can be turned into a program verification method known as *semi-proving* [8], which is an integration of testing, proving, and debugging. In semi-proving, symbolic inputs have been used to verify whether the program satisfies a given MR for either the *entire* input domain or selected paths in the program.

A simple program $Med(a, b, c)$ is given in [8] to illustrate the method. Its control flow is shown in Figure 3 (a). The program is expected to return the median of three real numbers a, b, and c. Obviously, the program is expected to observe the property $Med(\pi(a, b, c)) = Med(a, b, c)$ for any input (a, b, c) and any permutation $\pi(a, b, c)$. In fact, according to group theory [17], only two identities need to be verified, namely $Med(b, a, c) = Med(a, b, c)$ and $Med(a, c, b) = Med(a, b, c)$. These two identities are taken as the metamorphic relations.

Semi-proving verifies these metamorphic relations by applying the techniques of *global symbolic evaluation* [10, 11] followed by *constraint solving* [19]. Global symbolic evaluation is a technique that executes every possible path

of the program with symbolic inputs. Similar *symbolic analysis* techniques are also intensively studied in the area of parallelizing compilers [13, 14]. For certain classes of programs, we can either prove that they satisfy the prescribed MR for the entire input domain or identify *all* failure-causing inputs violating the MR. Furthermore, these failure-causing inputs will be expressed in constraint expressions to support debugging. For example, for the faulty program with a "missing path error" as shown in Figure 3 (b), the failure-causing inputs can be described by the expression $b < a < c$. As a result, the nature of the program defect can be better revealed [8]. In situations where it is too difficult or expensive to perform global symbolic evaluation, semi-proving can still be applied to verify selected paths rather than the entire input domain. In this way, the cost of global symbolic evaluation and constraint solving can be reduced [8].

3.5. Stronger MRs may not necessarily be better than weaker ones

For a given problem, usually more than one metamorphic relation can be identified. It is important to know how to select the most effective MR. For example, some properties are theoretically stronger than others. Are stronger properties necessarily better than weaker ones? Our preliminary study in [6] suggests that it is not necessarily the case.

A program $ShortestPath(G, u, v)$ is used as a case study in [6]. The parameter G is an undirected weighted graph represented by a matrix. The program searches for the shortest path between vertices u and v in G. For a non-trivial input, it will be very expensive to verify the correctness of the output. An experiment is conducted with 21 mutants as faulty programs. For each mutant, 1000 pairs of metamorphic test cases are randomly generated.

The experiment is as follows: The identity relation $ShortestPath(\pi(H), \boldsymbol{A'}, B') = ShortestPath(H, \boldsymbol{A}, B)$ is employed to produce a hierarchy of metamorphic relations, where H is a graph randomly generated; $\pi(H)$ is a permutation of H; $\boldsymbol{A}$ and B are different vertices in H; and $\boldsymbol{A'}$ and B' are vertices in the graph $\pi(H)$ corresponding to $\boldsymbol{A}$ and B, respectively. Suppose H consists of 10 vertices $(v_0, v_1, \ldots, v_9)$. Let $\tau_i(H)$ be a transposition of H that exchanges the positions of vertices v_0 and v_i, $i = 1, 2, \ldots, 9$. Then, the input matrices corresponding to the graphs $H, \tau_1(H), \tau_2(H), \ldots, \tau_9(H)$ will be different. Let $\pi_1(H), \pi_2(H), \ldots, \pi_9(H)$ be another set of permutations of H obtained by circularly shifting $(v_0, v_1, \ldots, v_9)$ left by 1 digit, 2 digits, ..., and 9 digits, respectively.

According to group theory [17], the compositions of the transpositions $\tau_1, \tau_2, \ldots, \tau_9$ can generate any other permutations of H. Hence, the metamorphic relations $\tau_1, \tau_2, \ldots, \tau_9$ as a whole are obviously stronger than the metamorphic relations $\pi_1, \pi_2, \ldots, \pi_9$. The experimental results in [6] show, however, that overall failure-revealing ability of the metamorphic relations $\pi_1, \pi_2, \ldots, \pi_9$ is much higher than that of $\tau_1, \tau_2, \ldots, \tau_9$. In addition, although π_1 is the strongest property among $\pi_1, \pi_2, \ldots, \pi_9$ because it can generate any other π_i for $i = 2, 3, \ldots, 9$, it actually demonstrates the lowest failure-causing ability. Instead, π_5 demonstrates the highest failure-causing ability, followed by π_4 and π_6, then followed by π_3 and π_7, and then π_2 and π_8. The worst are π_1 and π_9.

Thus, the results suggest that theoretically stronger metamorphic relations may not necessarily have a higher failure-revealing ability than weaker ones.

3.6. How to select useful metamorphic relations

In addition to permutation properties, other MRs have also been investigated in [6]. Among them, the seemingly simplest property $ShortestPath(H, \boldsymbol{A}, B) = ShortestPath(H, B, \boldsymbol{A})$ has demonstrated the highest failure-causing ability. The reasons for this have also been studied: In metamorphic testing, the program is run on a first input case and then on a follow-up input case. Although the two inputs are different, they are related by the MR. Hence, the two executions of the program should have both similarities and differences. It is found in [6] that the bigger the differences between two executions, the more effective is the MR in revealing program defects. Take the permutation property discussed in Section 3.5 as an example. Although the second test case is different from the first one, the program's search for the shortest path on the two test cases basically follows a similar execution sequence, that is, searching from the starting vertex $\boldsymbol{A}$ (and correspondingly, $\boldsymbol{A'}$) to the adjacent edges and vertices until B (and correspondingly, B') has been reached. Since the two input graphs are permutations of each other, their adjacent vertices and edges also correspond to each other. Hence, their searching sequences are similar. As a result, it is relatively more likely that the two executions will produce the same outcome. On the other hand, when the MR $ShortestPath(H, \boldsymbol{A}, B) = ShortestPath(H, B, \boldsymbol{A})$ is used, the starting and ending points in the second execution are swapped and hence the searching sequence is reversed: The program will start from B and search towards $\boldsymbol{A}$. In this way, the two execution sequences differ from each other greatly. Consequently, it is relatively more likely that the two executions will produce different outcomes if there is a fault in the program.

In short, it is suggested in [6] that, when selecting MR to test a given program, the algorithm and structure of the program should be taken into consideration. Metamorphic relations that can make the second execution *most different*

from the first execution are likely to achieve the best failure-revealing effect. It should be noted that the concept of "difference between two executions" has not been defined explicitly. It is a concept covering all aspects of program executions. For example, it may include the sequence of statements exercised, sequence of different values taken by program variables, and so on. Further research should be conducted to give more explicit guidelines.

4. Conclusion

In this paper, we have presented the basic concept of metamorphic testing and its applications in program testing, proving, and debugging. We have also highlighted several important issues that are critical to the fault-detection effectiveness of metamorphic testing. In addition to program verification, our semi-proving technique also supports debugging by generating constraint expressions for the failure-causing inputs. This kind of expression (such as $b < a < c$ for a program $p(a, b, c)$) are more informative and explicit than actual test cases (such as $a = 1.3$, $b = -2.8$, $c = 3.6$) in identifying defects, and may even give clues to the correction of the program. In terms of failure-revealing ability, we have observed that a stronger metamorphic relation is not necessarily better than a weaker metamorphic relation that can be derived from it. As future research, it will be interesting to find out the desirable characteristics of metamorphic relations that are good at revealing failures.

References

[1] B. Beizer, *Software Testing Techniques,* Van Nostrand Reinhold, New York, 1990.

[2] M. Blum and S. Kannan, "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.

[3] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems", *Journal of Computer and System Sciences,* 47(3), 1993, pp. 549–595.

[4] 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, Hong Kong, 1998.

[5] 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 Annual International Computer Software and Applications Conference (COMPSAC 2002)*, IEEE Computer Society Press, Los Alamitos, California, 2002, pp. 327–333.

[6] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou, "A case study on the selection of useful relations in metamorphic testing", paper in preparation.

[7] T.Y. Chen, F.-C. Kuo, Y. Liu, and A. Tang, "Metamorphic testing and testing with special values", In *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD2004)*, International Association for Computer and Information Science, Mt. Pleasant, Michigan, 2004.

[8] 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 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, ACM Press, New York, 2002, pp. 191–195.

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

[10] L.A. Clarke and D.J. Richardson, "Symbolic evaluation methods: implementations and applications", In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing,* North-Holland, Amsterdam, 1981, pp. 65–102.

[11] L.A. Clarke and D.J. Richardson, "Applications of symbolic evaluation", *Journal of Systems and Software,* 5(1), 1985, pp. 15–35.

[12] W.J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions,* Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[13] T. Fahringer and B. Scholz, "A unified symbolic evaluation framework for parallelizing compilers", *IEEE Transactions on Parallel and Distributed Systems*, 11(11), 2000, pp. 1105–1125.

[14] T. Fahringer and B. Scholz, *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization,* Volume 2628 of *Lecture Notes in Computer Science,* Springer, Berlin, 2003.

[15] C.F. Gerald and P.O. Wheatley, *Applied Numerical Analysis,* Addison Wesley, Reading, Massachusetts, 1999.

[16] A. Gotlieb and B. Botella, "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.

[17] M. Hall, Jr., *The Theory of Groups,* AMS Chelsea, Providence, Rhode Island, 1999.

[18] D. Hamlet, "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.

[19] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap, "The CLP(R) language and system", *ACM Transactions on Programming Languages and Systems,* 14(3), 1992, pp. 339–395.

[20] L.I. Manolache and D.G. Kourie, "Software testing using model programs", *Software: Practice and Experience,* 31(13), 2001, pp. 1211–1236.

[21] L.J. Morell, "A theory of fault-based testing", *IEEE Transactions on Software Engineering,* 16(8), 1990, pp. 844–857.

[22] G.J. Myers, *The Art of Software Testing,* Wiley, New York, 1979.

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