HKU CSIS Tech Report TR-2002-07

To appear in *Information and Software Technology* **44** (2002)

Fault-Based Testing Without the Need of Oracles [*] [†]

T. Y. Chen [‡]
School of Information Technology,
Swinburne University
of Technology, Australia

T. H. Tse *and* Zhiquan Zhou
Department of Computer Science
and Information Systems,
The University of Hong Kong

Abstract

There are two fundamental limitations in software testing, known as the reliable test set problem and the oracle problem. Fault-based testing is an attempt by Morell to alleviate the reliable test set problem. In this paper, we propose to enhance fault-based testing to alleviate the oracle problem as well. We present an integrated method that combines metamorphic testing with fault-based testing using real and symbolic inputs.

Keywords: Fault-based testing, metamorphic testing, oracle problem, symbolic execution.

1 Introduction

Program correctness has long been one of the most fundamental issues of computer science. Although program proving provides a formal means of verifying the correctness of programs, it suffers from the complexity and automation of the proofs. It is not easy even to prove the correctness of a relatively simple program. On the other hand, software testing is the most popular method used by practitioners to improve their confidence in the software product. There are, however, two recognized limitations in software testing, known as the reliable test set problem and the oracle problem. The concept of a *reliable test set* was originally proposed by Howden [20]: Suppose p is a program computing function f on domain D. A test set $T \subset D$ is *reliable* for p if $\big(\forall t \in T,\ p(t) = f(t)\big) \Rightarrow \big(\forall t \in D,\ p(t) = f(t)\big)$. In other words, the success of a reliable test set implies the program correctness. Howden points out, however, that an effective algorithm which generates a reliable test set for any given program cannot be constructed, unless the set covers the whole input domain. We refer to this limitation as the *reliable test set problem* or the *reliability problem*. Another deficiency in software testing is that, in some situations, testers are unable to decide whether $p(t) = f(t)$, that is, whether the result of the program under testing agrees with the expected result. This second limitation is known as the *oracle problem* [17, 29].

Since reliable test sets of finite sizes are not attainable in general, and test sets employed in practice must be of finite sizes, testers need practical means of evaluating such test sets with a view to selecting those with

[*]This research is supported in part by the Hong Kong Research Grants Council and the University of Hong Kong Committee on Research and Conference Grants.

[†]A preliminary version of this paper was presented at the 25th Annual International Computer Software and Applications Conference (COMPSAC 2001) [10].

[‡]All correspondence should be addressed to T. Y. Chen, School of Information Technology, Swinburne University of Technology, Hawthorn 3122, Australia. Email: tychen@it.swin.edu.au.

better performances. The mutation adequacy (or relative adequacy) criteria were introduced [5, 15, 16] to provide a realistic approach for determining whether a test set is relatively sufficient. It restricts the faulty programs to a smaller set, possibly finite in size. Such faulty programs can be differentiated from the original program by a test set that is also finite. Thus, suppose p is a program computing function f on domain D, and Q is a finite set of programs generated by slightly modifying the original program p. Each program $q \in Q$ such that $q \neq p$ is called a *mutant* of p. A test set $T \subset D$ is said to be *adequate for p relative to Q* if, $\forall$ programs $q \in Q$, $\big(\exists t \in D : q(t) \neq f(t)\big) \Rightarrow \big(\exists t \in T : q(t) \neq f(t)\big)$. The purpose of *mutation testing* is to generate a relative adequate test set T to differentiate all the mutants $q \in Q$ from the original program p. Mutation testing has been shown to be very powerful in revealing program faults both experimentally and analytically [16, 26, 27, 28]. This is because, as research into the "fault coupling effect" [18, 19, 25] demonstrated, "Complex faults are *coupled* to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults" [25].

In mutation testing, it is assumed that any set of mutants consists only of a finite number of programs, so that they can be killed by a finite test set. This constraint has been resolved by Morell using the concept of *fault-based testing* [22]. He refers to mutants as *alternate programs*, and a set of mutants as an *alternative set*, which may contain an infinite number of programs. Hence, fault-based testing "prove[s] the absence of infinitely many faults based on finitely many executions" [22]. To achieve this goal, the technique of symbolic execution [11, 12, 21] was used, and statements proclaiming the absence of certain types of faults were created and proved during the testing process. In this way, Morell combined program testing and proving in a unified methodology.

Given any input to a program, an *oracle* is a mechanism that specifies the expected outcome. We note that, in the above methodologies for alleviating the reliable test set problem, there is always an underlying assumption that a testing oracle exists. Testers check the execution results against the oracles to decide whether the program generates correct results on test cases. In some practical situations, however, an oracle is not attainable. This is known as the oracle problem. In numerical analysis, for example, it is often difficult to verify the results of calculations [17]. Weyuker [29] defined a program to be non-testable if "(1) there does not exist an oracle" or "(2) it is theoretically possible, but practically too difficult to determine the correct output." Moreover, in the theory of fault-based testing introduced by Morell, not only is the oracle for real output required, but the oracle for symbolic output is also demanded because it involves symbolic execution and symbolic output. Without an oracle, the above techniques will not work. In this paper, we propose an integrated approach that combines fault-based testing with metamorphic testing to alleviate the oracle problem. Our method is built on the techniques of symbolic execution [11, 12, 14, 21, 24].

In Section 2, we shall introduce the concepts in fault-based testing, and highlight the need of a testing oracle. In Section 3, we shall review testing techniques proposed by various researchers that can be carried out in the absence of an oracle. In particular, we shall introduce the metamorphic testing technique. In Section 4, we shall present our approach that integrates fault-based testing with metamorphic testing in order to alleviate the oracle problem in the former technique. We shall demonstrate through examples how real and symbolic inputs can be used to rule out prescribed faults in programs even if testing oracles are not available. In Section 5, we shall discuss how the method can be applied further. The final section will conclude the paper.

```
1:  INPUT (x, y);
2:  x = x * y + 3;
3:  OUTPUT (x * 2);
```

Figure 1: Program p for $f(x,\ y) = 2xy + 6$

```
1:  INPUT (x, y);
2′: x = x * y + F;
3:  OUTPUT (x * 2);
```

Figure 2: Program p'

2 Fault-Based Testing

There have been a lot of discussions on the purposes of software testing. Most people agree that testing cannot prove the correctness of a program [2]. Some people regard testing as an activity to look for bugs in a program, and therefore consider successful test cases, which fail to reveal errors, to be useless and a waste of time [23]. Others argue that successful test cases are useful and informative [6, 22]. Fault-based testing adopts the latter perspective and treats successful executions of a program as indications of the absence of some types of faults [22]. Fault-based testing therefore receives from a successful execution the information on the absence of certain types of faults. In some sense, mutation testing can be regarded as a special case of fault-based testing. A major difference is that the set of mutants eliminated by the former is finite whereas, by making use of symbolic executions, the set of alternate programs eliminated by fault-based testing can be infinite.

2.1 Fault-based testing with real input

Figure 1 shows a 3-line program p adapted from the first example in [22], which illustrates the technique of using one single *symbolic alternative* to represent infinitely many alternatives. The program is supposed to calculate a mathematical function $f(x,\ y) = 2xy + 6$. To ensure that there is no error with respect to the constant "3" in line 2, we assume that it is replaced by another constant "F", as shown in line $2'$ of program p' in Figure 2. "F" denotes all possible alternatives for the constant "3", and hence program p' represents infinitely many alternate programs for p.

Let $x = 5$ and $y = 6$ be a test case. The original program p will generate an output of 66, which can easily be verified to be correct against an oracle. By means of symbolic execution of program p', we obtain an output of $(30 + F) * 2$. Morell's goal is to find all the constants F such that program p' will produce the same result as the original program p. In other words, we must find all the values of F such that $(30 + F) * 2 = 66$. Solving the equation, we obtain $F = 3$. Hence, we have proved that the test case $(5,\ 6)$ distinguishes the original program p from all mutants constructed by replacing 3 in line 2 by any other constant values. Note that, to do the testing, an oracle is required for checking the correctness of the output of the original program.

```
     double ComputeArea ( ) {
     double a, b, incr, area, v;
1:   INPUT (a, b, incr); /* incr > 0 */
2:   v = a * a + 1;
3:   area = 0;
4:   while (a + incr <= b) {
5:          area = area + v * incr;
6:          a = a + incr;
7:          v = a * a + 1;
     }
8:   incr = b − a;
9:   if (incr >= 0) {
10:         area = area + v * incr;
11:         return area;
     }
     else
12:         ERROR (“illegal values for a and b!”);
     }
```

Figure 3: Program *ComputeArea*

2.2 Fault-based testing with symbolic input

The previous example illustrated the procedure of using a real input to eliminate a constant alternative in fault-based testing. Morell also demonstrated how to eliminate more complex alternatives such as variable substitution using symbolic inputs rather than real inputs. Figure 3 shows a sample program adapted from [22]. It calculates the area under the curve $x^2 + 1$ over the interval between a and b. Suppose the aim of the testing is to show the absence of errors in the assignment statement 3. Let the symbolic input be $a = A$, $b = B$, and $incr = I$ such that $B \geq A$ and $A + I > B$. Then the symbolic output produced by symbolic execution will be $(A*A+1)*(B-A)$. Note that, according to Morell's method, a *symbolic oracle* is required here to verify the correctness of the output.

Suppose we introduce a fault in the assignment statement 3:

$3'$: area = F; /* Should be “area = 0;” */

where F is a constant. Following the same execution path, we obtain an output of $F + (A*A+1)*(B-A)$. Morell's goal is to find all the constants F such that statement $3'$ will produce the same result as the original statement 3. Hence, we have $F + (A^2+1)*(B-A) = (A^2+1)*(B-A)$, which can be solved to give $F = 0$. Thus, statement $3'$ can only be exactly the same as statement 3.

Morell also proved that alternate programs would also be eliminated when F denoted a polynomial of a. In other words, if $F(a)$ denotes the set of all polynomials in a, then it can be proved that $F(a)$ can only be 0.

Furthermore, Morell introduced another error in statement 5. It is replaced by

$5'$: area = F; /* Should be “area = area + v * incr;” */

where F denotes a constant alternative. Let the symbolic input be $a = A$, $b = B$, and $incr = I$ such that $A + I \leq B$ and $A + 2I > B$. Using a similar procedure, it can be shown that $F = (A*A+1)*I$, thus contradicting the assumption that F is a constant. This proves that no constant substitution can be found for statement 5. By executing the loop twice, Morell further eliminated all alternate programs in which F could be a polynomial of the variables *area*, v, and *incr*. In other words, when the assignment fault introduced in statement 5 is a polynomial in the form $F(area,\ v,\ incr)$, it can be proved [22] that $F(x,\ y,\ z) = x + yz$, which is exactly the function computed in the original program.

Note again that these techniques have been based on the assumption that both the real and symbolic outputs can be verified against some oracles.

In Section 4, we shall present an approach to do fault-based testing without the need of oracles for real and symbolic outputs. Before doing this, let us review the oracle problem in more details.

3 Testing Without Oracles

In order to test a numerical program where an oracle is not available, a common approach is to make use of identity relations derived from theory. This technique was, for example, intensively used in Cody and Waite [13]. For instance, the identity $\cos(-x) = \cos(x)$ was used to test the program that supposedly compute the cosine function.

Weyuker [29] undertook a detailed study and introduced various approaches to test "non-testable programs" via static and dynamic properties of the functions being calculated. She gave an example of the testing of two programs that are supposed to compute the functions $f(x)$ and $f'(x)$, respectively, where f' is the derivative of f. From elementary results in Taylor series, we know that $f(x+\Delta) = f(x) + \Delta \times f'(x) + O(\Delta^2)$. Substituting $\Delta = 1,\ 0.1,\ 0.01,\ \ldots$ into the expression $f(x+\Delta) - (f(x) + \Delta \times f'(x))$, we can "see at a glance whether f' could be the derivative of f."

There is a closely related technique known as *data diversity*. It has been developed and advocated for fault tolerance, rather than fault detection, by Ammann and Knight [1]. Given an original input, the objective of data diversity is to provide alternate means of computing the same input using the same program. Such alternate inputs are basically "reexpressed" forms of the original input. Consequently, properties used in data diversity must also be identity relations.

Blum and Kannan [3] introduced the concept of a *program checker*, which is a program that probabilistically checks the correctness of the output of another program. An example is a checker for the graph isomorphism function, which employs the property that if G and H are not isomorphic, then G and permutations of H should not be isomorphic either. Blum et al. [4] extended the theory of the program checker into the theory of *self-testing / correcting*. Given a function f and a program p that implements f, a *self-tester T* for f is a probabilistic program. T estimates the error probability that $p(x) \neq f(x)$ for a random input x. A *self-corrector C* for f is also a probabilistic program. If it is known that program p calculates f correctly for sufficiently large amount of data on the input domain, then for any input x, C will make calls to p and return the value of $f(x)$ correctly with a high probability. Blum et al. introduced general techniques to construct self-tester / corrector for a variety of numerical functions. For example, the self-tester / corrector for integer multiplication functions essentially employs the distributive law $a \times (b + c) = a \times b + a \times c$. The self-tester / corrector for modular functions essentially employs the property that $(a+b) \bmod r = (a \bmod r + b \bmod r) \bmod r$.

More recently, a *metamorphic testing* (MT) method was proposed by Chen et al. [8]. It can be explained briefly as follows. Let f be a function to be programmed. Suppose R_f is some property of f that can be expressed as a relation among a series of the function's inputs $x_1, x_2, \ldots, x_n$, where $n > 1$, and their corresponding values $f(x_1),\ f(x_2), \ldots,\ f(x_n)$. This relation R_f is called a *metamorphic relation*. Consider the sine function, for instance. For any two inputs x_1 and x_2 such that $x_1 + x_2 = \pi$, we must have $\sin\ x_1 = \sin\ x_2$. This property is a metamorphic relation of the sine function and can be written formally as

$$R_{\sin} = \big\{(x_1,\ x_2,\ \sin\ x_1,\ \sin\ x_2)\ \big|\ x_1 + x_2 = \pi\ \ \rightarrow\ \ \sin\ x_1 = \sin\ x_2\big\}.$$

When there is no ambiguity, we can simply write the relation as

$$R_{\sin}:\ \ x_1 + x_2 = \pi\ \ \rightarrow\ \ \sin\ x_1 = \sin\ x_2.$$

Suppose p is a program that implements the function f. Let $p(x_1), p(x_2), \ldots, p(x_n)$ be the outputs of p corresponding to the inputs $x_1, x_2, \ldots, x_n$, respectively. In theory, p should satisfy all the properties of f, including metamorphic relations R_f. In practice, however, the relations R_f need to be converted into other metamorphic relations R_p more suitable for the implementation domain, by taking into account such implementation issues as rounding errors in floating-point arithmetic. MT proposes to check whether a program under test satisfies such metamorphic relations R_p. They are *necessary* (but not sufficient) conditions for the correctness of the program under test.

For example, suppose p implements the sine function. In theory, it should satisfy

$$R_p = \big\{(x_1,\ x_2,\ p(x_1),\ p(x_2))\ \big|\ x_1 + x_2 = \pi\ \ \rightarrow\ \ p(x_1) = p(x_2)\big\}.$$

In practice, when floating point arithmetic is involved, the inputs and outputs should satisfy an implementation-oriented metamorphic relation such as

$$R'_p = \big\{(x_1,\ x_2,\ p(x_1),\ p(x_2))\ \big|\ x_1 + x_2 = PI\ \ \rightarrow\ \ \left|\frac{p(x_2) - p(x_1)}{\min(|p(x_2)|,\ |p(x_1)|)}\right| < \varepsilon\big\}$$

or

$$R''_p = \big\{(x_1,\ x_2,\ p(x_1),\ p(x_2))\ \big|\ x_1 + x_2 = PI\ \ \rightarrow\ \ |p(x_2) - p(x_1)| < \varepsilon\big\},$$

where PI is the implemented value of π and ε is the acceptable error. For the ease of presentation, we shall simply use the form R_p in the examples in this paper. Readers are reminded that R'_p or R''_p may be used in the actual cases.

To verify this relation, two executions are needed in MT. The first input to p is a real number x_1, followed by a second input $x_2 = \pi - x_1$. Even if a testing oracle does not exist, MT can still be applied because it checks the relations among the inputs and outputs of more than one execution of the program, instead of checking a single result.

There is a similarity between MT and the earlier methods introduced in this section, in that all of them make use of some properties of the functions to check the program outputs. There are, however, differences between MT and the other methods in both practice and philosophy. In practice, metamorphic testing not only employs identity relations, but also makes use of inequalities. An example can be found in [9]. As for the other approaches described above, apart from a couple of examples on error bounds given by Weyuker, they all employ identity relations only. With regard to the philosophical aspect, consider the program checker as an example. Its ultimate goal is to estimate, through a probabilistic oracle, how likely the program output is correct for a given test case. Even though other test cases may be generated by the checker during the testing process, the fundamental goal does not change. On the other hand, it is not the prime objective of metamorphic testing to provide an alternate means of identifying a testing oracle to verify the correctness of a single output. Its intrinsic philosophy is that, when a test case selected according to some testing criteria does not reveal any failure, it still carries useful information, albeit implicitly. Thus, follow-up test cases can be used to check certain necessary properties of the program, irrespective of whether a testing oracle exists or not. If the necessary properties do not hold, the program is obviously incorrect. In this way, metamorphic testing is property-based and can be used along with any other test case selection strategies.

4 Integrating Fault-Based Testing with the Metamorphic Method

As introduced in Section 3, MT is a method that checks whether the program satisfies expected metamorphic relations. The latter is independent of the presence or otherwise of an oracle. MT can therefore be applied without the need of an oracle. In this section, we shall integrate MT and fault-based testing to alleviate the oracle problem, for both real and symbolic inputs.

4.1 Preliminary example

Similar to Morell's fault-based testing, our integrated method also allows two types of inputs: real and symbolic. We shall also use the mathematical function $f(x,\ y) = 2xy + 6$ and the 3-line program in Figure 1 as a preliminary example to illustrate our technique. From simple algebra, we find that the function satisfies the property $f(xy,\ f(x,\ y)) - f(x,\ y) = (2xy)^2 + 10xy$. [1] This can be expressed formally as a metamorphic relation as follows:

$$\begin{aligned} R_f \ = \ \big\{ \big((x_1,\, y_1),\ (x_2,\, y_2),\ f(x_1,\, y_1),\ f(x_2,\, y_2) \big) \ \big| \\ \big(x_2 = x_1 y_1 \ \wedge\ y_2 = f(x_1,\, y_1) \big) \ \rightarrow\ f(x_2,\, y_2) - f(x_1,\, y_1) = (2x_1y_1)^2 + 10x_1y_1 \big\}. \end{aligned}$$

Let p denote the program in Figure 1. The expected metamorphic relation for p is defined as

$$\begin{aligned} R_p \ = \ \big\{ \big((x_1,\, y_1),\ (x_2,\, y_2),\ p(x_1,\, y_1),\ p(x_2,\, y_2) \big) \ \big| \\ \big(x_2 = x_1 y_1 \ \wedge\ y_2 = p(x_1,\, y_1) \big) \ \rightarrow\ p(x_2,\, y_2) - p(x_1,\, y_1) = (2x_1y_1)^2 + 10x_1y_1 \big\}. \end{aligned}$$

For the test case $(x_1,\ y_1) = (5,\ 6)$, the program p produces "66" as output. Suppose, for the sake of argument, that this program does not have a known oracle.[2] We continue to generate the next test case as suggested by the metamorphic relation R_p. Thus, we obtain $x_2 = x_1y_1 = 5 \times 6 = 30$, and $y_2 = p(x_1,\ y_1) = 66$. For this test case, the program yields $p(30,\ 66) = 3966$. We need to verify whether the two test results together satisfy the expected relation R_p. Indeed, $p(x_2,\ y_2) - p(x_1,\ y_1) = 3966 - 66 = 3900$, and $(2x_1y_1)^2 + 10x_1y_1 = (2 \times 5 \times 6)^2 + 10 \times 5 \times 6 = 3900$. Hence, the relation R_p is fulfilled.

Suppose we introduce an assignment fault F into the program, as shown in Figure 2. This faulty program, which we shall denote by p', produces $2F + 60$ by the symbolic execution of the same initial test case $(x_1,\ y_1) = (5,\ 6)$. Taking the metamorphic relation R_p into consideration, we choose a second symbolic test case $(x_2 = 5 \times 6 = 30)$ and $(y_2 = p'(5,\ 6) = 2F + 60)$. After symbolic execution, p' yields the output $(30 \times (2F + 60) + F) \times 2 = 122F + 3600$. Our goal is to find the value(s) of F for which p' satisfies the

[1]Many properties of f can be identified as metamorphic relations. This is but one example.

[2]We use this simple but artificial example to illustrate the procedure behind metamorphic testing. Genuine examples where no oracle exists will be given in Sections 4.2 and 4.3.

```
double Power (double u, double v) {
double uMinusOne, numerator, lnTerm, result;
int i;
if (v = = 0)
    result = 1;
else {
    if ((int)v = = v) && (v > 0)) {
        result = 1;
        for (i = 1; i <= v; i++)
            result = result * u;
    }
    else {
    /* ln (u) = ln (1 + (u − 1)) = (u − 1) − 1 / 2 * (u − 1)ˆ2 + 1 / 3 * (u − 1)ˆ3 − . . . */
        i = 1;
        uMinusOne = u − 1;
        numerator = uMinusOne;
        lnTerm = uMinusOne;
        result = uMinusOne;
        while (fabs (lnTerm) > 1e−16) {
        /* “fabs” is a floating point function that returns the absolute value */
        /* 1e−16 = 10ˆ{−16} */
            i++;
            numerator = (−1) * numerator * uMinusOne;
            lnTerm = numerator / i;
            result = result + lnTerm;
        }
        result = exp(v * result);
    }
}
return result;
}
```

Figure 4: Program *Power*

expected relation R_p. Thus, substituting into R_p, we establish $p'(x_2, y_2) - p'(x_1, y_1) = (2x_1y_1)^2 + 10x_1y_1$, giving $(122F + 3600) - (2F + 60) = (2 \times 5 \times 6)^2 + 10 \times 5 \times 6$. Solving the equation, we obtain $F = 3$, which is the only value that F can take. This means that all the alternate programs constructed by replacing 3 with other constants have been eliminated. This result coincides with that obtained by conventional fault-based testing in [22]. A fundamental difference is that we have applied the MT technique without referring to a testing oracle.

In the next two sections, we shall further describe how fault-based testing can be achieved in the absence of an oracle using real and symbolic inputs, respectively.

4.2 Fault-based testing with real input in the absence of an oracle

Consider the program *Power* in Figure 4. Given two real numbers u and v as input, the program computes the value of u^v. This is done in three ways:

(i) If v is zero, then obviously $u^v = 1$.

(ii) Otherwise, if v is a positive integer, then u^v can be found by multiplying u by itself the appropriate number of times.

(iii) Otherwise, u^v is computed by the mathematical expression $e^{v \ln(u)}$.

The main task of our testing lies with part (iii).

Consider statement 11 in the program. Suppose we introduce a fault in the statement 11, of the form

11′ : result = F; /* Should be "result = uMinusOne;" */

Our goal here is to ensure that any constant alternative is impossible. Assume the contrary. We would like to find all possible constants F such that the erroneous statement $11'$ would pass the test without being detected.

Since it is not straightforward to verify the result of this example against an oracle, especially for large numbers, we shall make use of the metamorphic testing method. A typical property of the exponential function is $u^v \times u^v = (u \times u)^v$. Hence, the program should satisfy the metamorphic relation

$$Power(u, v) \times Power(u, v) = Power(u \times u, v).$$

Let $u = 0.5400128$ and $v = 3.9$ be a test case. The original program will generate $Power(0.5400128, 3.9) = 9.0443177318673 \times 10^{-2}$, and $Power(0.5400128 * 0.5400128, 3.9) = 8.1799683234969 \times 10^{-3}$. Since $(Power(0.5400128, 3.9))^2 = 8.1799683234969 \times 10^{-3} = Power(0.5400128 * 0.5400128, 3.9)$, the metamorphic relation is satisfied.

Now consider the program with the symbol "F" in statement $11'$. Let us call this program $Power'$. After symbolic execution, we obtain the symbolic output

$$Power'(0.5400128, 3.9) = e^{3.9 \times [F + \Sigma_{i=2}^{43} (-1)^{i-1} (0.5400128-1)^i / i]}.$$

Hence,

$$Power'(0.5400128, 3.9) \times Power'(0.5400128, 3.9) = e^{2 \times 3.9 \times [F + \Sigma_{i=2}^{43} (-1)^{i-1} (0.5400128-1)^i / i]}.$$

On the other hand, for the test case $(0.5400128 \times 0.5400128,\ 3.9)$, the output of the symbolic execution is

$$Power'(0.5400128 * 0.5400128, 3.9) = e^{3.9 \times [F + \Sigma_{i=2}^{94} (-1)^{i-1} (0.5400128 \times 0.5400128-1)^i / i]}.$$

Hence, according to the metamorphic relation, we should have

$$e^{2 \times 3.9 \times [F + \Sigma_{i=2}^{43} (-1)^{i-1} (0.5400128-1)^i / i]} = e^{3.9 \times [F + \Sigma_{i=2}^{94} (-1)^{i-1} (0.5400128 \times 0.5400128-1)^i / i]}.$$

Solving the equation, we obtain $F = -2.1158822416384 \times 10^{-1}$.

```
/* Program Trig calculates the value of sin x if "isSin" is true.
Otherwise, it calculates the value of cos x. */

       double Trig (double x, bool isSin) {
       int index;
       double term, sum;
       /* Initialize for cos x */
1:     term = 1;
2:     index = 1;
3:     if (isSin) {  /* Initialize for sin x */
4:            term = x;
5:            index = 2;
       }
6:     sum = term;
7:     while (fabs (term) > 1e−16) {
8:            term = term * (−1) * x * x / (index * (index + 1));
9:            sum = sum + term;
10:           index = index + 2;
       }
11:    return sum;
       }
```

Figure 5: Program *Trig*

Let $u = 0.7309782$ and $v = 9.16$ be another test case. Through the same procedure, we can deduce that $F = -7.2372728875240 \times 10^{-2}$. Even if rounding errors are taken into consideration, those two values of the same constant F obviously contradict each other. As a result, no constant alternative is possible for statement $11'$. In other words, we have proved that the metamorphic test cases $(u, v) = (0.5400128, 3.9)$, $(0.5400128 \times 0.5400128, 3.9)$, $(0.7309782, 9.16)$, and $(0.7309782 \times 0.7309782, 9.16)$ distinguish the original program *Power* from every mutant constructed by replacing *uMinusOne* in statement 11 by any constant value.

4.3 Fault-based testing with symbolic input in the absence of an oracle

Let us consider a further example as shown in Figure 5. The program *Trig* accepts as inputs a real number x and a Boolean parameter *isSin*. It calculates $\sin x$ when *isSin* is true and $\cos x$ when *isSin* is false. Except for special cases such as $x = 0$ and $x = \pi/2$, there is no easy way to verify the outputs of the program, unless we check them against the outputs from yet another program.

Suppose statement 4 is replaced by an alternative:

$4'$:   term = F;   /* Should be "term = x;" */

where F is a function of x in the form $F = a \times x^n$ such that a is a real number constant and n is a non-negative integral constant.

As explained earlier, it is not easy to verify the program against an oracle, so that we cannot apply Morell's fault-based testing method to eliminate the alternative. Instead, let us test the program using the property $\sin^2 x + \cos^2 x = 1$. The metamorphic relation in the implementation domain can be written as

$$R_p: \quad (p(x, \text{true}))^2 + (p(x, \text{false}))^2 = 1. \tag{1}$$

Consider a symbolic input $x = S$ with $isSin = \text{true}$. By symbolic execution of the alternate program containing statement $4'$, the output is

$$Trig\,(S, \text{true}) = F * (1 - S^2/6 + S^4/120 - \dots).$$

Hence, we have

$$\begin{aligned}(Trig\,(S, \text{true}))^2 &= F^2 * (1 - S^2/6 + S^4/120 - \dots)^2 \\ &= (a * S^n)^2 * (1 - S^2/6 + S^4/120 - \dots)^2 \\ &= (a^2 * S^{2n}) * (1 - (1/3) * S^2 + (2/45) * S^4 - \dots) \\ &= a^2 * S^{2n} - (1/3) * a^2 * S^{2*(n+1)} + (2/45) * a^2 * S^{2*(n+2)} - \dots\end{aligned} \tag{2}$$

For the same symbolic input $x = S$ with $isSin = \text{false}$, we obtain a second symbolic output of the program, thus:

$$Trig\,(S, \text{false}) = 1 - S^2/2 + S^4/24 - S^6/720 + \dots$$

As a result,

$$(Trig\,(S, \text{false}))^2 = 1 - S^2 + (1/3) * S^4 - (2/45) * S^6 + \dots$$

Based on equation (1), we should have

$$(Trig\,(S, \text{true}))^2 + (Trig\,(S, \text{false}))^2 = 1$$

and therefore

$$a^2 * S^{2n} - (1/3) * a^2 * S^{2*(n+1)} + (2/45) * a^2 * S^{2*(n+2)} - \dots = S^2 - (1/3) * S^4 + (2/45) * S^6 - \dots \tag{3}$$

As we know, n is a non-negative integral constant. If $n = 0$, then there will be a constant term a^2 on the left-hand side of identity (3), while the minimum degree of the right-hand side will be 2. This is obviously a contradiction. If $n \geq 2$, then all the terms on the left-hand side of the identity will have a degree ≥ 4, while there will be a term of degree 2 on the right-hand side. This again will be a contradiction. Thus, the only possible value of n is 1. Since $n = 1$, if we equate the coefficients of like terms on both sides of the identity, we obtain $a^2 = 1,\ -(1/3) * a^2 = -1/3,\ (2/45) * a^2 = (2/45),\ \dots$, which can be solved to give $a = \pm 1$.

In this way, F can only be x or $-x$. Of course, $F = x$ is just the original statement 4 in program $Trig$. Hence, we need only study $F = -x$. To discard this alternative, we need to employ another metamorphic relation. For example, we note that $\sin x = -\cos(\pi/2 + x)$. The metamorphic relation in the implementation domain can be written as

$$x_2 = \pi/2 + x_1 \ \rightarrow\ p(x_1, \text{true}) + p(x_2, \text{false}) = 0. \tag{4}$$

We apply the technique introduced in Section 4.2 to eliminate the alternative $F = -x$ using real input. Let $PI = 3.1415926535898$ and let the input be $x = 1.2$. The original program with $F = x$ in statement 4 will produce $Trig(1.2, \text{true}) = 0.93203908596723$, and $Trig(PI/2+1.2, \text{false}) = -0.93203908596723$. Hence, $Trig(1.2, \text{true}) + Trig(PI/2+1.2, \text{false}) = 0$ and the metamorphic relation (4) is satisfied.

We continue to test the alternate program with statement 4 replaced by "term = $-$x;". Let us call it $Trig'$. For the same input $x = 1.2$, the results are $Trig'(1.2, \text{true}) = -0.93203908596723$ and $Trig'(PI/2 + 1.2, \text{false}) = -0.93203908596723$. Hence, $Trig'(1.2, \text{true}) + Trig'(PI/2 + 1.2, \text{false}) = -1.86407817193446$. Obviously, this alternate program does not satisfy the expected metamorphic relation (4). In this way, the alternative $F = -x$ is eliminated.

In this section, we have illustrated our technique of using symbolic input (possibly combined with real input) to eliminate prescribed faults. The example also demonstrated the power of combining different metamorphic relations in testing. It shows that, by making use of more than one metamorphic relation, different types of faults may be revealed. In other words, different metamorphic relations may have different fault-detection capabilities for different types of faults.

We must concede, however, that our method may not be foolproof in terms of program correctness. This issue will be further discussed in the next section.

5 Discussions

In the previous examples, the prescribed types of faults have been totally eliminated. This may not be possible in some real life situations. Having said that, we shall illustrate via an example in this section how our method may still be applied in such circumstances.

The program *Trap* as shown in Figure 6 is adapted from [11]. It calculates the approximate area under the curve $f(x)$ for the interval between $x = a$ and $x = b$. Suppose statement 12 is replaced by an alternate statement

$12'$: area = area + (k_1 * yOld + k_2 * yNew) / 2.;
/* Should be "area = area + (yOld + yNew) / 2.;" */

where k_1 and k_2 are any constants. Before testing, let us first identify a metamorphic relation. Suppose $G(x) = F(x) + C$, where C is a positive constant. From elementary calculus, we know that $Trap(G, A, B, N, Error) = Trap(F, A, B, N, Error) + C \times |B - A|$ when $N \geq 1$. We execute the program using the symbolic input $(F, A, B, N, Error)$, where F is any function, $A > B$, and $N = 1$. The statements $(1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 9, 14, 15, 16, 17)$ will be traversed, generating the symbolic output

$$Trap(F, A, B, N, Error) = (k_1 * F(A) + k_2 * F(B))/2 * (A - B).$$

Running the program again using the symbolic input $(G, A, B, N, Error)$, we obtain

$$Trap(G, A, B, N, Error) = (k_1 * (F(A) + C) + k_2 * (F(B) + C))/2 * (A - B).$$

According to the metamorphic relation, therefore, we establish an equation

$$(k_1 * (F(A) + C) + k_2 * (F(B) + C))/2 * (A - B) = (k_1 * F(A) + k_2 * F(B))/2 * (A - B) + C \times (A - B).$$

After simplification, we obtain $(k_1 + k_2) * C = 2C$, thus giving $(k_1 + k_2) = 2$. If k_1 and k_2 are possible integers, then both of them can only take the value of "1", which is just the original value. Otherwise, we

```
/* Program Trap implements the trapezoidal rule to find the approximate area under the curve f(x) between
x = a and x = b. The computation uses n intervals of size |b − a| / n each. The variable “error” will be set
to “true” when n < 1. */

float Trap (float (*f)(float), float a, float b, int n, bool &error) {
     float area;
     float h; /* interval */
     float x;
     float yOld; /* value of f(x − h) */
     float yNew; /* value of f(x) */

1:   if (n < 1)
2:       error = true;
     else {
3:       error = false;
4:       area = 0;
5:       if (a != b) {
6:           h = (b − a) / n;
7:           x = a;
8:           yOld = (*f)(x);
9:           while ((a > b && x > b) || (a < b && x < b)) {
10:              x = x + h;
11:              yNew = (*f)(x);
12:              area = area + (yOld + yNew) / 2.;
13:              yOld = yNew;
                 }
14:          area = area * h;
15:          if (a > b)
16:              area = −area;
             }
         }
17:      return area;
         }
```

Figure 6: Program *Trap*

can still eliminate all pairs of k_1 and k_2 such that $k_1 + k_2 \neq 2$. This example shows that, even in situations where our method cannot exactly identify the fault, our technique is still useful because it greatly narrows down the range of possible faults.

The examples cited so far in this paper are numerical programs. It should be noted that our approach can also be applied to non-numerical ones. Consider, for instance, a program *ShortestPath* that implements the shortest path problem. The program accepts a graph G and two nodes A and B, and then outputs all the shortest paths from A to B. Apart from simple graphs, it is expensive to verify whether the outputs are correct. In this case, metamorphic testing can be applied as follows: Randomly select an element P from the output of $ShortestPath(G, A, B)$. P is one of the shortest paths from A to B. Randomly select a node C in this path P. Then, run the program to compute $ShortestPath(G, A, C)$ and $ShortestPath(G, C, B)$. A metamorphic relation is, “there exist an element of $ShortestPath(G, A, C)$ and an element of $ShortestPath(G, C, B)$ that can be combined to form the path *P*." If this metamorphic relation is not satisfied, the program must contain a fault. Another metamorphic relation is that a different permutation of the input graph *G* should produce the same output. Fault-based testing techniques can be applied to such non-numerical metamorphic relations. Furthermore, the concepts of attributive equivalence and observational equivalence have been introduced in [7] for the testing of object-oriented programs. These concepts of equivalence can also be used as non-numerical metamorphic relations in fault-based testing.

6 Conclusion

In this paper, we have looked into the oracle problem in fault-based testing. We have found that, by integrating metamorphic testing with fault-based testing, alternate programs can be eliminated even if there is no testing oracle. We have presented techniques of using real and symbolic inputs.

When compared with other fault-based testing approaches that rely on testing oracles, our approach requires additional efforts in identifying metamorphic relations and running the program more than once. Obviously, whenever a testing oracle is available, it should be used to check the output. Nevertheless, there are many situations where a testing oracle cannot be found. Our method does provide an innovative solution in such circumstances.

References

[1] P. E. Ammann and J. C. Knight, Data diversity: an approach to software fault tolerance, *IEEE Transactions on Computers* **37** (4) (1988) 418–425.

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

[3] M. Blum and S. Kannan, Designing programs that check their work, *Journal of the ACM* **42** (1) (1995) 269–291.

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

[5] T. A. Budd, Mutation analysis: ideas, examples, problems and prospects, in: B. Chandrasekaran and S. Radicchi, eds., *Computer Program Testing* (North-Holland, Amsterdam, 1981) 129–148.

[6] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu, Application of metamorphic testing in numerical analysis, in: *Proceedings of the IASTED International Conference on Software Engineering* (*SE '98*) (ACTA Press, Calgary, Canada, 1998) 191–197.

[7] H. Y. Chen, T. H. Tse, and T. Y. Chen, TACCLE: a methodology for object-oriented software testing at the class and cluster levels, *ACM Transactions on Software Engineering and Methodology* **10** (1) (2001) 56–109.

[8] 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).

[9] 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).

[10] T. Y. Chen, T. H. Tse, and Z. Zhou, Fault-based testing in the absence of an oracle, in: *Proceedings of the 25th Annual International Computer Software and Applications Conference* (*COMPSAC 2001*) (IEEE Computer Society Press, Los Alamitos, California, 2001) 172–178.

[11] L. A. Clarke and D. J. Richardson, Symbolic evaluation methods: implementations and applications, in: B. Chandrasekaran and S. Radicchi, eds., *Computer Program Testing* (North-Holland, Amsterdam, 1981) 65–102.

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

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

[14] G. Colman, P. Andreae, and L. Groves, Program analysis by symbolic execution and generalization, in: C. Rattray and G. Robert, eds., *The Unified Computation Laboratory: Modelling, Specifications, and Tools* (Clarendon Press, Oxford, 1992) 367–380.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Hints on test data selection: help for the practicing programmer, *IEEE Computer* **11** (4) (1978) 34–41.

[16] R. A. DeMillo and A. J. Offutt, Constraint-based automatic test data generation, *IEEE Transactions on Software Engineering* **17** (9) (1991) 900–910.

[17] M.-C. Gaudel, Testing can be formal, too, in: *Proceedings of the 6th International Joint CAAP/FASE Conference on Theory and Practice of Software Development* (*TAPSOFT '95*) (Lecture Notes in Computer Science **915**, Springer-Verlag, Berlin, 1995) 82–96.

[18] K. S. How Tai Wah, Fault coupling in finite bijective functions, *Software Testing, Verification and Reliability* **5** (1) (1995) 3–47.

[19] K. S. How Tai Wah, A theoretical study of fault coupling, *Software Testing, Verification and Reliability* **10** (1) (2000) 3–45.

[20] W. E. Howden, Reliability of the path analysis testing strategy, *IEEE Transactions on Software Engineering* **SE-2** (3) (1976) 208–215.

[21] W. E. Howden, Symbolic testing and the DISSECT symbolic evaluation system, *IEEE Transactions on Software Engineering* **SE-3** (4) (1977) 266–278.

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

[23] G. J. Myers, *The Art of Software Testing* (John Wiley, New York, 1979).

[24] A. J. Offutt and E. J. Seaman, Using symbolic execution to aid automatic test data generation, in: *Systems Integrity, Software Safety, and Process Security: Proceedings of the 5th Annual Conference on Computer Assurance* (*COMPASS '90*) (IEEE Computer Society Press, Los Alamitos, California, 1990) 12–21.

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

[26] A. J. Offutt and S. D. Lee, An empirical evaluation of weak mutation, *IEEE Transactions on Software Engineering* **20** (5) (1994) 337–344.

[27] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, An experimental determination of sufficient mutant operators, *ACM Transactions on Software Engineering and Methodology* **5** (2) (1996) 99–118.

[28] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs against Errors* (John Wiley, New York, 1998).

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