Postprint of article in *Proceedings of the 25th Annual International Computer Software and Applications Conference* (*COMPSAC 2001*), IEEE Computer Society, Los Alamitos, CA, pp. 172–178 (2001)

Fault-Based Testing in the Absence of an Oracle [*] [†]

T. Y. Chen
Swinburne University of Technology
Melbourne, Australia

T. H. Tse [‡] and Zhiquan Zhou
The University of Hong Kong
Pokfulam, Hong Kong

Abstract

Although testing is the most popular method for assuring software quality, there are two recognized limitations, 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 address 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

There are two accepted ways of verifying the correctness of a program: proving and testing. The former suffers from the difficulty of the proofs. It is not easy even to prove the correctness of a relatively simple program. Although testing is the most popular method for assuring software quality, there are two recognized limitations, known as the reliable test set problem and the oracle problem. Much research in software testing has adopted the definition of the *reliable test set* originally given by Howden [16]: 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's result shows, however, that a reliable test set of a finite size is not attainable in general. It is called the *reliable test set problem* or the *reliability problem*. The other fundamental limitation in software testing is that, in some situations, testers are unable to decide whether $p(t) = f(t)$, that is, whether the program outcome is correct. This is known as the *oracle problem* [14, 27].

Because of the reliable test set problem, it is necessary to seek other approaches to defining the test set quality. The concept of an adequate test set was introduced [6, 12, 13]: Let p be a program computing function f on domain D. A test set $T \subset D$ is *adequate for* p if, $\forall$ programs q, $\big(\exists t \in D : q(t) \neq f(t)\big) \Rightarrow \big(\exists t \in T : q(t) \neq f(t)\big)$. Here, a different perspective of testing is taken: Instead of directly showing program correctness, an adequate test set is aimed at uncovering errors in every faulty program, which is more intuitive. Unfortunately, it was also shown that there is no effective procedure to generate an adequate test set or to decide whether a given test set is adequate [6, 13]. To alleviate this problem, the mutation adequacy (or relative adequacy) criteria were introduced [5, 12, 13], which limit the faulty programs to a set of finite size. Thus, suppose p is a program computing function f on domain D, and Q is a finite set of programs. A test set $T \subset D$ is *adequate for p relative to Q* if, $\forall$ programs $q \in Q$, $\big(\exists t \in D : q(t) \neq f(t)\big) \Rightarrow \big(\exists t \in T : q(t) \neq f(t)\big)$. In *mutation testing*, each program $q \in Q$ such that $q \neq p$ is called a *mutant* of p. The purpose of mutation testing is to generate a mutation adequate test set T to kill all the mutant programs in Q. Mutation testing was shown to be very powerful in revealing program faults both experimentally and analytically [13, 23, 24, 26]. Since the "fault coupling effect" is very rare [15, 22], test cases that detect simple faults can also detect more complex faults.

Morell [19] further developed the theory of *fault-based testing*. In mutation testing, the set of mutants must be

[†] 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.

[‡] All correspondence should be addressed to Prof. T. H. Tse, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk.

finite. On the other hand, this set (called the set of *alternative programs* by Morell) can be infinite in fault-based testing. Hence, fault-based testing "prove[s] the absence of infinitely many faults based on finitely many executions" [19]. To achieve this goal, the technique of symbolic execution [9, 17] 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.

It should be noted that in all the above methodologies for alleviating the reliable test set problem, there is always an underlying assumption that a testing oracle exists. An *oracle* is any mechanism specifying the expected execution result of a program on input data. Testers check the program output against the oracle to decide whether the program has resulted correctly 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 [14]. Weyuker [27] defined a program to be non-testable if there is no oracle or if the oracle is too difficult to be obtained in practice. 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 results. Without an oracle, all the above techniques will not work. In this paper, we shall propose an integrated approach that combines symbolic testing with metamorphic testing (to be introduced in Section 3) to alleviate the oracle problem for both real and symbolic outputs. Our method is built upon the techniques of symbolic execution [10, 21] and constraint solving [13, 28].

2. Fault-Based Testing

As we know, the correctness of a program cannot be proved by means of testing [2]. There have been different perspectives regarding the purpose of software testing. Some regard testing to be a means of revealing program faults, and therefore consider successful test cases, which fail to reveal errors, to be useless and a waste of time [20]. Others argue that successful test cases are useful and informative [7, 19]. Fault-based testing adopts the latter perspective and treats successful executions of a program as indications of the absence of some types of faults [19]. 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 alternative programs eliminated by fault-based testing can be infinite.

```
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'

(a) Fault-based testing with real input

The key of symbolic testing is to represent "infinitely many alternatives by a single *symbolic alternative*" [19]. The 3-line program P, as shown in Figure 1, is the first example given by Morell to illustrate the idea of symbolic testing. The program is supposed to calculate $f(x,\ y) = 2xy + 6$. 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 output 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 carry out testing, an oracle is required for checking the correctness of the output of the original program.

(b) Fault-based testing with symbolic input

The above example shows how to eliminate the constant substitution in fault-based testing with real input data. Morell also illustrated how to eliminate more complex alternatives such as variable substitution. To achieve this goal, symbolic inputs instead of real inputs were accepted. Figure 3 shows a sample program taken from [19]. It calculates the area below the graph of the function $x^2 + 1$ over the interval between

```
     Program ComputeArea (input, output);
     var a, b, incr, area, v: real;
     begin
1:   read (a, b, incr); { incr > 0 }
2:   v := a * a + 1;
3:   area := 0;
4:   while a + incr <= b do
          begin
5:        area := area + v * incr;
6:        a := a + incr;
7:        v := a * a + 1;
          end;
8:   incr := b − a;
9:   if incr >= 0 then
          begin
10:       area := area + v * incr;
11:       writeln ('area by rectangular method: ', area);
          end
     else
12:       writeln ('illegal values for a = ', a, ' and b = ', b);
     end.
```

Figure 3. Program ComputeArea

a and b. Suppose the aim of the symbolic 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 denotes 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;" }

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, the author obtained $F = (A * A + 1) * I$, thus contradicting the above. This proves that no constant substitution can be found for statement 5. By executing the loop twice, Morell further eliminated all alternative 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 multinomial in the form $F(area, v, incr)$, it can be proved [19] that $F(x, y, z) = x + yz$, which is exactly the function computed in the original program.

We would like to point out again that these techniques have been based on the assumption that both the real and symbolic outputs can be verified against some oracles. Otherwise, "in the case when only real outputs and not symbolic outputs can be verified", it is "necessary to resort to algebraic testing" [19]. In algebraic testing [18], however, an oracle for the real output is anyway needed, and restrictions on the programs to which the algebraic testing technique can be applied is so strict that the technique is more a result in theory than an approach in practice.

In Section 4, we shall present an approach to carry out fault-based testing in the absence of an oracle for real and symbolic outputs. Before doing this, let us review the oracle problem in more details.

3. Testing Without an Oracle

Weyuker [27] 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 computed 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 formula $f(x+\Delta) - (f(x) + \Delta \times f'(x))$, we can "see at a glance whether f' could be the derivative of f."

Following up with her work, many other approaches were proposed to assist automatic verification of the correctness of the testing result without the involvement of a human oracle [1, 3, 4, 7]. The basic idea of these approaches were also to employ some mathematical properties of the function from the theory and check whether the program being tested adheres to these properties. Blum et al. [4] 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 this relation should also be satisfied between G and permutations of H.

Blum et al. [3] 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. The underlying concept of this technique is also to make use of mathematical properties of function f. 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 $(a+b) \bmod r = (a \bmod r + b \bmod r) \bmod r$.

Recently, a *metamorphic testing* (MT) method [7, 8] was proposed, which can be explained as follows. Let $x_1, x_2, \ldots, x_n$ be different inputs to a function f. Suppose that, given some relation R among $x_1, x_2, \ldots, x_n$, their results $f(x_1)$, $f(x_2), \ldots, f(x_n)$ must satisfy some mathematical property R_f. In formal terms, we have

$$R(x_1, x_2, \ldots, x_n) \Rightarrow R_f\big(f(x_1), f(x_2), \ldots, f(x_n)\big) \qquad (1)$$

For example, if the function in question is "sine", then for any two inputs x_1 and x_2 such that $x_1 + x_2 = \pi$, we must have $\sin(x_1) = \sin(x_2)$. In formal terms,

$$(x_1 + x_2 = \pi) \Rightarrow \big(\sin(x_1) = \sin(x_2)\big). \qquad (2)$$

Let P be a program implementation of the function f. Suppose $P(x_1), P(x_2), \ldots, P(x_n)$ are the outputs for different inputs $x_1, x_2, \ldots, x_n$. If the program is correct, the inputs and outputs must obviously satisfy a mathematical criterion similar to (1) above, namely

$$R(x_1, x_2, \ldots, x_n) \Rightarrow R_f\big(P(x_1), P(x_2), \ldots, P(x_n)\big)$$

This property is known as a metamorphic relation. It is a *necessary* condition for the correctness of P. MT proposes to check whether a program under test satisfies such metamorphic relations. For example, for a program P that computes the "sine" function, two executions are needed to verify the metamorphic relation (2). The first input to P is any real number x, and the second input is $\pi - x$. The two outputs are expected to be equal. Even if a testing oracle does not exist, MT can still be applied because it checks the relations among the outputs of several executions of the program, instead of checking a single execution 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 other methods in both practice and philosophy. In practice, the function properties employed by the previous testing methods, such as checkers and self-testers / correctors, are equalities. In other words, although those properties can also be regarded as metamorphic relations, they are all related with equality. For example, the checker for the graph isomorphism function [4] makes use of the relation that if isomorphism$(G, H) =$ FALSE, then isomorphism$(G, H') =$ FALSE, where H' is any permutation of H. Another example is the self-tester / corrector for integer multiplication functions [3], which essentially employs the equality $a \times (b+c) = a \times b + a \times c$. In metamorphic testing, on the other hand, the metamorphic relations may include, but are not limited to, equality. MT employs a wider range of properties. For example, if $f(x)$ is a monotone increasing function, then a metamorphic relation of f can be that "$f(x) > f(y)$ if $x > y$".

There also exists a difference in the purpose of testing between other methods and MT. For example, the ultimate goal of the program checker is to verify the correctness of a single output. Even though other test cases may be generated by the checker during the testing process, the fundamental goal does not change. For metamorphic testing, on the other hand, the ultimate goal is not to verify the correctness of a single output, but is to verify whether the program being tested possesses the expected properties on all the input data over the input domain.

4. Integrating Fault-Based Testing with the Metamorphic Method

As introduced in Section 3, metamorphic testing (MT) is a method that checks whether the program satisfies expected metamorphic relations. MT can therefore be performed in the absence of an oracle. In this section, we shall integrate MT and the symbolic testing to alleviate the oracle problem for 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 use the 3-line program in Figure 1 as a preliminary example to illustrate our technique for real input. By simple algebra, we find that $f(x, y) + f(-x, y) = 12$. In formal terms, the metamorphic relation is

$$(x_1 = -x_2 \text{ and } y_1 = y_2) \Rightarrow \big(f(x_1, y_1) + f(x_2, y_2) = 12\big).$$

For the test case $(x, y) = (5, 6)$, the original program in Figure 1 produces "66" as output. Suppose that this program does not have a known oracle. [1] We continue to

[1] 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.

generate next test case based on the metamorphic relation. Thus, we obtain $(x,\ y) = (-5,\ 6)$ as the second test case. On this input the program yields -54. Now it needs to be verified the two test cases satisfy the metamorphic relation. Since $66 + (-54) = 12$, the test is passed.

Let us now introduce the assignment fault F into the program, as shown in Figure 2. For the same initial test case $(x,\ y) = (5,\ 6)$, the program produces $2F + 60$ by symbolic execution. For the second test case $(x,\ y) = (-5,\ 6)$, the program yields $2F - 60$ by symbolic execution. Our goal is to solve for the value(s) of F with which the program satisfies the expected relation $f(x, y) + f(-x, y) = 12$. In other words, we must solve for F in the equation $(2F + 60) + (2F - 60) = 12$. We obtain $F = 3$. This means that all the alternate programs constructed by replacing 3 by other constants have been eliminated using the metamorphic test cases $(5,\ 6)$ and $(-5,\ 6)$. This is the same conclusion obtained by the conventional fault-based testing in [19]. A fundamental difference is that by applying the MT technique this time, a testing oracle is not needed any more.

The above example alone may not be sufficient to convince readers, because there is actually an oracle in Morell's program, although we have chosen not to use it. We shall describe in the next two sections how fault-based testing can be achieved in the absence of an oracle, using real and symbolic inputs.

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

A program *ComputePower* is given in Figure 4. Pascal is used for the purpose of consistency in this paper, since the other programs quoted from [19] are all in this language. Our method works equally well in other programming languages such as C.

Given two real numbers a and b as input, the program computes the value of a^b. This is done in three ways:

(i) If b is zero, then obviously $a^b = 1$.

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

(iii) Otherwise, a^b is computed by the mathematical formula $e^{b \ln(a)}$.

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

We have to deal with large number arithmetic in many applications, such as in cryptography software [25]. We cannot rely on the standard programs for the logarithm or exponential function, and hence there is a need to write mathematical functions of our own. Errors may therefore

```
     Program ComputePower (input, output);
     var i, n: integer;
     var a, b, check, logarithm, mainFactor, plusMinus, power,
         term, x : real;
     begin
1:   read (a, b);
2:   if b = 0 then
3:       power := 1
     else begin
4:       n := trunc (b);
5:       check := n;
6:       if (b > 0) and (check = b) then
             begin
7:           power := 1;
8:           for i := 1 to n do
9:               power := power * a;
             end
         else begin
         { ln (a) = ln (1 + x) = x − 1 / 2 * x ^ 2 + 1 / 3 * x ^ 3 − ... }
10:          x := a − 1;
11:          i := 1;
12:          plusMinus := 1;
13:          mainFactor := x;
14:          term := x;
15:          logarithm := x;
16:          while abs (term) > 0.00000001 do
                 begin
17:              i := i + 1;
18:              plusMinus := −1 * plusMinus;
19:              mainFactor := x * mainFactor;
20:              term := plusMinus * mainFactor / i;
21:              logarithm := logarithm + term;
                 end;
22:          power := exp(b * logarithm);
             end;
         end;
23:  writeln (a:10:6, ' ^ ', b:10:6, ' = ', power:10:6);
     end.
```

Figure 4. Program ComputePower

be introduced as a result. Furthermore, there are no obvious oracles for verifying the results of these functions. If there were alternative functions that provide such results to us, then there would not have been a need for writing such functions in the first place. One of the standard techniques in testing complex functions is to consider special cases, such as when the inputs are integers. Unfortunately, it will not work in this example because separate techniques (i) and (ii) have been implemented for such special cases.

For the ease of presentation in this paper, we shall *not* use complex techniques for the calculation of logarithm in *ComputePower*. In spite of this, we hope that the rationale behind the lack of an oracle has already been explained clearly to readers.

Consider statement 15 in the program. Assume that we can introduce a fault in the statement 15, of the form

```
15′ :  logarithm := F;    { Should be "logarithm := x;" }
```

Our goal is to find all the constants F (if any) such that statement $15'$ will produce the same result as the original statement 15.

In the absence of an oracle, we would like to make use of the metamorphic testing method. A property in the mathematical function a^b is that $a^b \times a^b = (a \times a)^b$. For real number arithmetic, where we cannot guarantee exact equality, the property can be written as an inequality

$$\left| \frac{a^b \times a^b - (a \times a)^b}{(a \times a)^b} \right| < 10^{-5}.$$

Let $a = 0.8$ and $b = 2.1$ be a test case. After symbolic execution of the program with the symbol "F" in statement $15'$, we obtain the following symbolic output:

$$0.8^{2.1} = e^{2.1 \times [F + \sum_{i=2}^{11} (-1)^{i-1} (0.8-1)^i / i]}.$$

Hence,

$$0.8^{2.1} \times 0.8^{2.1} = e^{2 \times 2.1 \times [F + \sum_{i=2}^{11} (-1)^{i-1} (0.8-1)^i / i]}.$$

On the other hand, for the test case $(0.8 \times 0.8,\ 2.1)$, the output of the symbolic execution is

$$(0.8 \times 0.8)^{2.1} = e^{2.1 \times [F + \sum_{i=2}^{16} (-1)^{i-1} (0.8 \times 0.8-1)^i / i]}.$$

Since

$$\left| \frac{0.8^{2.1} \times 0.8^{2.1} - (0.8 \times 0.8)^{2.1}}{(0.8 \times 0.8)^{2.1}} \right| < 10^{-5},$$

we find that any constant F in statement $15'$ can only have a value of -0.04000 ± 0.00001.

Let $a = 0.3$ and $b = 4.2$ be another test case. By a similar procedure, we can show that any constant F in statement $15'$ can only have a value of -0.49000 ± 0.00001.

These two results for the same constant F obviously contradict each other. Hence, no constant can be found for statement $15'$. In other words, we have proved that the metamorphic test cases $(a,\ b) = (0.8,\ 2.1)$, $(0.8 \times 0.8,\ 2.1)$, $(0.3,\ 4.2)$, and $(0.3 \times 0.3,\ 4.2)$ distinguish the original *ComputePower* program from all mutants constructed by replacing x in statement 15 by any other constant values.

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 calculates the exponential e^x for any input x. Unless we are given another program that computes exactly the same function, we cannot easily find an oracle, except for the trivial test cases $x = 0$ and $x = 1$.

Suppose statement 6 is replaced by an alternative

```
6′ :  term := F * term / i;
      { Should be "term := x * term / i;" }
```

where F is a constant. Then, using the test cases $x = 0$ and $x = 1$, we can easily apply Morell's fault-based testing to show that F will take two different values, hence leading to a contradiction. This proves that no constant substitution can be found for x in statement 6.

```
      Program ComputeExponent (input, output);
      var i: integer;
      var x, term, exponent: real;
      begin
1:    read (x);
      { exponent = exp (x) = 1 + x + x ^ 2 / 2! + x ^ 3 / 3! + ... }
2:    i := 1;
3:    term := 1;
4:    exponent := 1;
5:    while abs (term) > 0.00000001 do
          begin
6:        term := x * term / i;
7:        exponent := exponent + term;
8:        i := i + 1;
          end;
9:    writeln ('e ^ ', x:10:6, ' = ', exponent:10:6);
      end.
```

Figure 5. Program ComputeExponent

On the other hand, in the absence of a testing oracle, it will be difficult to apply the original fault-based testing method to eliminate other alternative statements such as

```
6″ :  term := x * x * term / i;
      { Should be "term := x * term / i;" }
```

Instead, let us test the program using the metamorphic relation $e^x \times e^x = e^{2x}$. Consider a symbolic input $x = X$. By symbolic execution of the alternative program containing statement $6''$, the output value of *exponent* is

$$1 + X * X + X * X * X * X / 2 + \ldots$$

Hence, we have

$$e^x \times e^x = 1 + 2 * X * X + 2 * X * X * X * X + \ldots$$

and

$$e^{2x} = 1 + 4 * X * X + 8 * X * X * X * X + \ldots$$

Thus, we have a contradiction, which means that the alternative statement $6''$ is impossible. Similar procedures will enable us to prove that, in general, alternative statements of the form

```
term := x * x * ... * x * term / i;
      { Should be "term := x * term / i;" }
```

where x is multiplied n times for $n \geq 2$, are impossible.

We must concede that our integrated method may not be foolproof. For example, given an alternate statement of the form

```
term := F * x * term / i;
      { Should be "term := x * term / i;" }
```

where F is a constant, we cannot use the same procedure to prove that F must have a value of 1. Other properties of the

exponent function will have to be identified to achieve this effect.

5. 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, alternative programs can be eliminated in the absence of oracles. We have presented techniques of using real and symbolic inputs.

We note, however, that the method will depend on the properties of the functions under test, and is therefore only a partial solution to the oracle problem, which in general is unsolvable. As pointed out by Offutt [21], "the fact that partial results for theoretically intractable problems are valuable is a fortunate observation for software engineering, since many software engineering problems do have negative theoretical properties."

We have only illustrated our approach using relatively simple programs in this paper. Further studies will be required for more complex program structures. Future research will also include the development of automated software testing tools to support our method.

References

[1] L. M. Adleman, M.-D. Huang, and K. Kompella. Efficient checkers for number-theoretic computations. *Information and Computation*, 121 (1): 93–102, 1995.

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

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

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

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

[6] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18 (1): 31–45, 1982.

[7] 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*), pages 191–197. ACTA Press, Calgary, Canada, 1998.

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

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

[11] D. Coward and D. Ince. *The Symbolic Execution of Software: the SYM-BOL System*. Chapman and Hall, London, 1995.

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

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

[14] 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*), volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Berlin, 1995.

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

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

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

[18] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10 (1): 53–66, 1978.

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

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

[21] 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*), pages 12–21. IEEE Computer Society, Los Alamitos, California, 1990.

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

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

[24] 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): 99–118, 1996.

[25] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, Upper Saddle River, New Jersey, 1999.

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

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

[28] J. Zhang. Specification analysis and test data generation by solving Boolean combinations of numeric constraints. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software* (*APAQS 2000*), pages 267–274. IEEE Computer Society, Los Alamitos, California, 2000.