Metamorphic Testing and Testing with Special Values

Tsong Yueh Chen, Fei–Ching Kuo[*], Ying Liu and Antony Tang
School of Information Technology
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
Email: {tchen, dkuo, yliu, atang}@it.swin.edu.au

Abstract

The problem of testing programs without test oracles is well known. A commonly used approach is to use special values in testing but this is often insufficient to ensure program correctness. This paper demonstrates the use of metamorphic testing to uncover faults in programs, which could not be detected by special test values. Metamorphic testing can be used as a complementary test method to special value testing. In this paper, the sine function and a search function are used as examples to demonstrate the usefulness of metamorphic testing. This paper also examines metamorphic relationships and the extent of their usefulness in program testing.

1. Introduction

Program testing has been one of the major challenges in establishing the correctness of a program. Theoretically, though there are ways to prove that a program is correct, it is not practical to do so for even moderately complex programs [4, 6]. A common and practical way to ensure program correctness is through program testing. Program testing often involves in selecting suitable inputs as test cases, executing the program and verifying results against expected results.

Ideally, for any input to the program, there is a mechanism to verify the correctness of the output. Suppose p is a program computing function f on domain D, and a test set T with test inputs t are used to test the program p. The mechanism of verifying the expected result is referred to as a test oracle but test oracles are not always available [7]. They may be difficult and expensive to obtain or they may be absent altogether. Another limitation in program testing is what test data

and how much of a test set should be used in order to sufficiently verify that a program is working correctly. A test set $T \subset D$ is said to be reliable for p if $(\forall t \, \epsilon \, T, p(t) = f(t)) \Rightarrow (\forall t \, \epsilon \, D, p(t) = f(t))$. The testing of program p with test set T is said to be reliable when T can reliably detect that $p(t) \neq f(t)$ if $p(t)$ contains a fault [5]. In reality, the input domain can be infinitely large and it is not easy to determine what test values should be used and definitely not practical to test the entire input domain. This limitation is referred to as reliable test set problem.

If test oracles are not available for testing, the verification of program results may resort to the use of special test values instead. Special test values are test values in which their expected results are well known and can be used to verify the program. For instance, the majority of results from a sine program are not immediately verifiable and only special test values with well known expected outputs can be used for verification. Some of the well known values are sin $\pi \, / \, 3$ and sin $\pi \, / \, 6$ etc.. Obviously, the use of special test values in testing has the inherent issue of the reliability problem, that is, the set of special test values used in testing may not be sufficient to reveal any program errors.

In this paper, we propose to use metamorphic testing to provide additional test cases to testing with special values. Recently, metamorphic testing has been proposed to ensure program correctness through verifying the correctness of some properties for the program [1]. Special value testing is used as a mean to illustrate and contrast with metamorphic testing in this paper and as such other testing approaches are not considered.

Section 2 of this paper provides some background on metamorphic testing. Section 3 contains two examples to demonstrate how metamorphic testing can be used effectively to complement special value testing. The testing of the sine function is described in section 3.1. The testing of a program that makes use of a combination

---

[*] Corresponding author

of binary and linear search is described in section 3.2. The conclusion is presented in section 4.

2. Background of metamorphic testing

Metamorphic testing was proposed to overcome some of the inherent problems that are present in testing software without test oracles [2, 3]. Metamorphic testing involves the verification of a metamorphic relation R of function f, of which p is an implementation. The concept of metamorphic relation R has a number of characteristics. Firstly, metamorphic relationships are necessary properties of the target function f. If any of these properties does not hold during testing, the program p is faulty. Secondly, a metamorphic relation is a relationship among the inputs and outputs involving multiple executions of p. In other words, program p must appear in the relation more than once. For instance, sine function has a well known property where for any inputs $x1$ and $x2$, if $x1 + x2 = \pi$, then $sin(x1) = sin(x2)$. The sine function is used twice with different parameters in the execution of the sine program. As a reminder, the testing of the relationship does not require verification of actual outputs of $sin(x1)$ and $sin(x2)$.

Although special value testing provides a way to test program correctness, there is no theoretical ground why special value testing is sufficient in ensuring program correctness. Metamorphic testing complements special value testing by its ability to test the necessary properties of the function. Moreover, the test data set used by metamorphic testing is augmented by using random test values.

3. Metamorphic Testing Examples

The examples used in this paper follow the general approach where a) a set of special values and expected test outputs are selected to be used in the testing; b) a fault is introduced to a test program, called a mutant program; c) metamorphic relations for the function are defined; d) perform metamorphic testing as well as special value testing.

Examples of metamorphic testing shown in this paper make use of the special test values to test the metamorphic relations. In addition, random test values are also used to extend the test data set and provide additional test cases for metamorphic testing.

3.1. Metamorphic Testing of Sine Function

Consider a program p which implements the sine function, $sin(\theta)$. The sine function f has a number of well known special test values that can be used in the testing of p. Test data and corresponding expected results are expressed in form of $(\theta, sin(\theta))$. Special values used in the testing are the following elements $\{(0, 0), (\pi/6, 1/2), (\pi/4, 1/\sqrt{2}), (\pi/3, \sqrt{3}/2), (\pi/2, 1)\}$. The selected list of special values have well known expected results between 0 and $\pi/2$. In order to distinguish between special value testing and property testing using metamorphic relationships, we restrict ourselves to selecting special test values from the range of 0 to $\pi/2$.

The program p (as shown in Figure 1) is an excerpt of the correct program for computing the sine function. A fault is introduced to program p to obtain faulty program fp. The example below shows the fault that has been introduced in fp. Faulty program fp is used to

```
1 double sin(double x, unsigned int qoff)
2 {
3        short a1;
4        a1=_Dtest(&x);
5        switch (a1)
6        {
7        case NAN:
8                errno = EDOM;
9                return (x);
10       case 0:
11               return (qoff ? 1.0 : 0.0);
12       case INF:
13               errno = EDOM;
14               return (_Nan._D);
15       default:
16               {
17               double g;
18               long quad;
19
20               if ( x < -HUGE_RAD|| HUGE_RAD < x) {
21                       g = x/twopi;
22                       _Dint(&g,0);
23                       x -= g * twopi;
24                       }
25               g=x*twobypi;
26               quad = (long)(0 < g ? g + 0.5 : g - 0.5);
27               qoff += (unsigned long)quad & 0x3;
28               g = (double)quad;
29               g = (x - g * c1) - g*c2;
30               if ((g <0.0 ? -g :g) < _Rteps._D) {
31                       if (qoff & 0x1)
32                               g = 1.0;
33                       }
34               else if (qoff & 0x1)
35                       g = _Poly(g*g,c,7);
36               else
37                       g *= _Poly(g*g,s,7);
38               return (qoff & 0x2 ? -g :g);
39               } /* default */
40       } /* case stmnt */
41 } /* sin() */
```

Figure 1. Program p to compute sine function

demonstrate that using special test values are inadequate for testing the sine function. The fault is a simple variation of the existing program statement and could have been made when constructing the program. Although there are numerous faults that could be introduced into the program, some of the faults produce errors that can easily be detected using special test values in testing and hence has little value in terms of demonstration. The following faulty program, however, produces errors that cannot be detected through the use of the defined special values and hence is considered a suitable candidate for demonstration purpose.

```
27   qoff = (unsigned long)quad & 0x1; // Error Statement
```

Since metamorphic relations are the necessary properties which the function f must hold, the following list of metamorphic relations should hold in the testing of fp if fp is working correctly.

R_{sin1} : $sin(x) = sin(x + 2\pi)$
R_{sin2} : $sin(x) = -sin(x + \pi)$
R_{sin3} : $-sin(-x) = sin(x)$
R_{sin4} : $sin(x) = sin(\pi - x)$
R_{sin5} : $sin(x) = -sin(2\pi - x)$
R_{sin6} : $sin(x) + sin(y) + sin(z) - sin(x+y+z) = 4 * sin((x+y)/2) * sin((x+z)/2) * sin((y+z)/2)$
R_{sin7} : $sin^2(x) + sin^2(\pi/2 - x)=1$
R_{sin8} : $sin(3 * x) = 3 * sin(x) - 4 sin^3(x)$
R_{sin9} : $sin(x)^2 - sin(y)^2 = sin(x + y) * sin(x - y)$
R_{sin10} : $sin(5 * x) = 16 * sin^5(x) + 5 * sin(3 * x) - 10 * sin(x)$

Table 1 shows the summary results for testing fp using special test values and metamorphic testing. The first column lists the input test data set that includes special test values as well as random test values. The second column, $fp(x) = expected\ result$, is the result of testing fp using special values to verify against expected results. All other columns R_{sini} are results of testing the metamorphic relations. In R_{sin6}, y and z values used are 2.35e+00 and 9.25e+00 respectively. The value of y in R_{sin9} is 2.35e+00.

Metamorphic testing does not require verification of the output of the sine function, that is sin(θ). The result is a verification of the inherent relationship to yield *true* or *false* as a result of examining the relationship R_{sini} by multiple executions of program p. Since a metamorphic relation should always hold true for the given input domain D where test data set $T \subset D$, therefore the program is faulty if any of the metamorphic tests fails. An entry of **T** in the table indicates that the relation holds true and **F** indicates that the relation does not hold true Computations of $fp(x)$ using random test values as inputs cannot be verified using special value testing. The results are listed to show that had these random values been used, the fault in $fp()$ could not have been revealed. From the test results (shown in Table 1), a number of very useful observations can be made. When special test values are used to test the faulty program fp, all tests yield the expected results. In other words, the fault that has deliberately been seeded in the faulty program fp cannot be detected using the list of special values. But through metamorphic testing, errors can be uncovered using special values as test input. Metamorphic testing can be carried out using random test values in addition to special test values. Testers may freely select inputs randomly to test relation R_{sini} without having to have expected results for sin(θ). In a sense, metamorphic testing provides a *self-test* mechanism based on the inherent metamorphic relationship. A single metamorphic relation may not sufficiently test fp, for instance, R_{sin1} alone cannot detect

.

Table 1. Results from metamorphic testing of sine program fp

fp Test Results											
	$fp(x) =$ expected result	R_{sin1}	R_{sin2}	R_{sin3}	R_{sin4}	R_{sin5}	R_{sin6}	R_{sin7}	R_{sin8}	R_{sin9}	R_{sin10}
Special Test Values											
0	T	T	T	T	T	T	F	T	T	F	T
π/6	T	T	F	T	F	T	F	T	T	T	F
π/4	T	T	F	F	F	T	F	T	F	T	F
π/3	T	T	F	F	T	F	F	T	F	T	F
π/2	T	T	F	F	T	F	F	T	F	F	F
Random Test Values											
1.71E-05	T	T	F	T	F	T	F	T	T	F	T
1.00E+00	T	T	F	F	T	F	F	T	F	T	F
2.13E-05	T	T	F	T	F	T	F	T	T	F	T
8.24E+00	T	T	F	F	T	F	F	T	T	F	F
1.29E-01	T	T	F	T	F	T	F	T	T	T	T
7.83E-01	T	T	F	T	F	T	F	T	T	T	F

any error using any of the special values and random values. R_{sin6} detects errors with all test cases. It uses the sine function seven times in the relation and the repeated invocations of the same function with different parameters provide a higher chance of uncovering program faults since each invocation with a different parameter could have different execution paths. Four metamorphic relations, namely R_{sin6}, R_{sin8}, R_{sin9} and R_{sin10}, make use of the sine function more than 2 times and they have consistently uncovered faults in program fp. The other six metamorphic relations R_{sin} make use of the sine function exactly 2 times, two of them did not uncover any fault and the other four have uncovered faults in program fp. The implication is that a metamorphic relation with more invocations of the function is more likely to detect errors.

On the other hand, given a specific fault, the ability of each metamorphic relation to detect the error is still depended on the nature of the fault and the test cases. Metamorphic testing uses a black box testing approach and it is not known before testing is performed as to what type of errors it might detect, therefore it is useful to perform metamorphic testing with all available metamorphic relations.

Finally, metamorphic testing has the flexibility to use random test values as input and not be restricted to only using special values. The use of random test values has the advantage over derived special values in that random test values do not have any implied property within the test data and are therefore more likely to avoid the possible issue of testing the same program path.

3.2. Metamorphic Testing of a Search Program

Consider a program p which implements function f that searches for an integer in an ordered data structure T. The input x to the function f is the integer value to search for, and the output of the function is -1 if x is not found, or when x is found, the program returns the starting and ending positions of x in T that is represented in the form of $(y1, y2)$. There are two inputs to test the program p. T is a set of integers sorted in ascending order. The integers contained in T can be duplicated. An input integer x is the test value used in the search.

A set of special values are defined for the testing of fp. The selection of special values to test fp is based on using particular isolated test values so that the test values would test critical points of the program [8]. The following are the criteria for choosing the special test values. (a) first data element; (b) last data element; (c) non-existent element; (d) all elements in T are identical; (e) T contains a single element; (f) all elements in T are distinct. Table 2 contains special values that are used in the testing. Program p uses a binary search method to search for the input x and then use linear search to find the starting and ending positions of x. This is listed in Figure 2. *BinSearch()* is used in the first call to find the existence of x and if found, the position of x is then passed to *GetRange()* to get the starting and ending positions of x.

To demonstrate the error detection capabilities of both special value testing and metamorphic testing, a fault is introduced to program p to become a faulty program fp as shown below.

```
28 *i_max = *i_max + 2; //ERR
```

Based on the nature of the function, the following metamorphic relations R_{si} are defined and they should hold true for the search function f.

R_{s1}: T is an ordered list containing integers in ascending order,
$\forall\ x_1\ \epsilon\ T : f(x_1, T)\ = (y1, y2)$ where $y1$ is the starting position of x_1 and $y2$ is the ending position of x_1 or $y1 = -1$ if $x_1 \notin T$,
$\forall\ x_2 = x_1 + 1\ \epsilon\ T : f(x_2, T)\ = (y3, y4)$ where $y3$ is the starting position of x_2 and $y4$ is the ending position of x_2 or $y3 = -1$ if $x_2 \notin T$,
then $(y1 = -1) \vee (y3 = -1) \vee ((y2 + 1) = y3)$

R_{s2}: T is an ordered list containing integers in ascending order,

Table 2. Special values used in testing of search program *fp*

Special Test Values	Test Value x	Test Value T	Expected Test Results
a) x is at beginning of T	-1	T_1 contains 900000 integers starting with element {-1} and ending with element {85963}	(0, 0)
b) x is at end of T	85963	T_1 contains 900000 integers starting with element {-1} and ending with element {85963}	(899999, 899999)
c) x does not exist in T	1000000	T_1 contains 900000 integers starting with element {-1} and ending with element {85963}	(-1)
d) all elements in T are identical	1	T_2 contains 5 elements of {1,1,1,1,1}	(0, 4)
e) T contains a single element	1	T_3 contains 1 element of {1}	(0, 0)
f) all elements in T are distinct	1	T_4 contains 5 element of {1,2,3,4,5}	(0, 0)

```
1 int BinSearch(int x, int i_start, int i_end)  // perform a
binary search of a number x over a range
2 {
3 int   i_mid;
4
5     if (i_start > i_end)
6         return -1;
7     else {
8         i_mid = (int)floor((i_start + i_end)/2);
9         if (i_array[i_mid] == x)
10             return(i_mid);
11         if (i_array[i_mid] > x)
12             BinSearch(x, i_start, i_mid - 1);
13         else
14             BinSearch(x, i_mid + 1, i_end );
15     }
16
17 } /* BinSearch */
18
19 int GetRange(int i, int *i_min, int *i_max)    // linear
search for a min to max range given a location i
20 {
21 int    i_val;
```

```
22
23     *i_min = *i_max = i;
24     //   search for upper limit
25     while (*i_max < i_arraysz) {
26         i_val = *i_max;
27         if (i_array[i] == i_array[i_val+1])
28             *i_max = *i_max + 1;
29         else
30             break;
31     }
32
33     //     search for minimum
34     while (*i_min > 0) {
35         i_val = *i_min;
36         if (i_array[i] == i_array[i_val-1])
37             *i_min = *i_min - 1;
38         else
39             break;
40     }
41     return 0;
42
43 } /* GetRange */
```

Figure 2. Program p to perform search on duplicated numbers

T' is $\{z\} \cup T$ where $z <$ head (T) and head(T) denotes the first element of T,

$\forall x \in T : f(x, T) = (y1, y2)$ where $y1$ is the starting position of x and $y2$ is the ending position of x or $y1 = -1$ if $x \notin T$,

$\forall x \in T'$ and $x \neq z : f(x, T') = (y3, y4)$ where $y3$ is the starting position of x and $y4$ is the ending position of x or $y3 = -1$ if $x \notin T'$,

then $(y1 = y3 = -1) \vee ((y3 = y1 + 1) \wedge (y4 = y2 + 1))$

The above metamorphic relations R_{si} are used in the testing of faulty program fp. A summary of the results are shown in Table 3. An entry of **T** in the table indicates that the relation holds true and **F** indicates that the relation does not hold. The results returned by the program are also listed in the table. In the second column, random test values cannot be used in the testing of $fp(x)$ due to the lack of expected test results, and hence they are reported as **N/A**. The faulty program fp satisfies all the test cases using special test values. Metamorphic testing cannot detect any errors using special test values either. After some analysis of the fault and the test cases, it is found that this particular fault is not revealed because it is data sensitive and works correctly with this set of special test values.

Metamorphic testing is then performed using randomly generated x and randomly generated T to check the validity of fp. The results indicate that out of the tests performed, R_{s1} detects 2 failures and R_{s2} detects 6 failures. $fp(x, T)$ cannot use random x and T in its testing because there is a lack of expected results. Arguable a tester could inspect T and design more test cases in addition to

the special test values. With careful inspection of T_1, it is found that when $x = 3$, the start and end positions of x are (8, 12) and when $x = 5$, the positions are (15, 15). Using these values to test fp will show that fp still works correctly. The expected results of $fp(x)$ when $x = 3$ and 5 are obtained by manually examining T_1. For more expected results to be obtained from a large data structure such as T_1 to be used in program verification, it would require developing another program which is not effective.

Metamorphic testing of R_{s1} and R_{s2}, on the other hand, does not require any expected results of fp for verification because they check the fundamental logic of fp. With random test values, Six out of seven tests of R_{s2} fail and two test cases fail using R_{s1} which indicates that R_{s2} seems to be more sensitive to this particular fault than R_{s1}. A closer examination of the results show that R_{s2} always fails in the condition $(y4 = y2 + 1)$, this information hints at the nature of the fault and would be useful for debugging purpose. With the ability to generate additional random test values for testing, more tests could be conducted to augment the test cases of R_{s1} and R_{s2},

4. Conclusion

This paper demonstrates the use of metamorphic testing as a complement to special value testing. Metamorphic testing can reveal faults where special value testing cannot. Both example programs used in this paper lack test oracles and hence special value testing is used. Special value testing has the deficiency that it is not possible to determine whether a special value test set is

Table 3. Metamorphic test results of faulty search program *fp*

fp Test Results			
	fp(x) = expected result	R_{s1}	R_{s2}
Special Test Values			
x is at beginning of T_1; *x = -1*	**T** : *fp*(-1, T_1) = (0, 0)	*fp*(-1, T_1) = (y1, y2) = (0, 0) *fp*(0, T_1) = (y3, y4) = (1, 2) **T** : ((y2 + 1) = y3)	*fp*(-1, T_1) = (y1, y2) = (0, 0) *fp*(-1, T_1') = (y3, y4) = (1, 1) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
x is at end of T_1; *x = 85963*	**T** : *fp*(85963, T_1) = (899999, 899999)	*fp*(85963 , T_1) = (y1, y2) = (899999, 899999) *fp*(85964, T_1) = y3 = -1 **T** : (y3 = -1)	*fp*(85963, T_1) = (y1, y2) = (899999, 899999) *fp*(85963, T_1') = (y3, y4) = (900000, 900000) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
x is not in T_1; *x = 1000000*	**T** : *fp*(1000000, T_1) = -1	*fp*(1000000 , T_1) = y1 = -1 *fp*(1000001, T_1) = y3 = -1 **T** : (y1 = -1) ∨ (y3 = -1)	*fp*(1000000,T_1)= y1 = -1 *fp*(1000000, T_1') = y3 = -1 **T** : (y1 = y3 = -1)
All elements in T_2 are identical ; *x = 1*	**T** : *fp(1, T_2) = (0, 4)*	*fp*(1 , T_2) = (y1, y2) = (0, 4) *fp*(2, T_2) = y3 = -1 **T** : (y3 = -1)	*fp*(1 , T_2) = (y1, y2) = (0, 4) *fp*(1, T_2') = (y3, y4) = (1, 5) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
T_3 contains 1 element ; *x = 1*	**T** : *fp(1, T_3) = (0, 0)*	*fp*(1 , T_3) = (y1, y2) = (0, 0) *fp*(2, T_3) = y3 = -1 **T** : (y3 = -1)	*fp*(1 , T_3) = (y1, y2) = (0, 0) *fp*(1, T_3') = (y3, y4) = (1, 1) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
All elements in T_4 are distinct; x = 1	**T** : *fp(1, T_4) = (0, 0)*	*fp(1, T_4)* = (y1, y2) = (0, 0) *fp*(2, T_4) = (y3, y4) = (1, 1) **T** : ((y2 + 1) = y3)	*fp(1, T_4)* = (y1, y2) = (0, 0) *fp(1, T_4')* = (y3, y4) = (1, 1) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
Random Test Values			
3, T_1	N/A	*fp*(3, T_1) = (y1, y2) = (8, 12) *fp*(4, T_1) = (y3, y4) = (13, 15) **T** : ((y2 + 1) = y3)	*fp*(3, T_1) = (y1, y2) = (8, 12) *fp*(3, T_1') = (y3, y4) = (9, 14) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
5, T_1	N/A	*fp*(5, T_1) = (y1, y2) = (15, 15) *fp*(6, T_1) = (y3, y4) = (16, 16) **T** : ((y2 + 1) = y3)	*fp*(5, T_1) = (y1, y2) = (15, 15) *fp*(5, T_1') = (y3, y4) = (16, 16) **T** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
348, T_1	N/A	*fp*(348, T_1) = (y1, y2) = (3141, 3146) *fp*(349, T_1) = (y3, y4) = (3146, 3146) **F** : ((y2 + 1) = y3)	*fp*(348, T_1) = (y1, y2) = (3141, 3146) *fp*(348, T_1') = (y3, y4) = (3142, 3146) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
3256, T_1	N/A	*fp*(3256, T_1) = (y1, y2) = (33751, 33765) *fp*(3257, T_1) = (y3, y4) = (33766, 33771) **T** : ((y2 + 1) = y3)	*fp*(3256, T_1) = (y1, y2) = (33751, 33765) *fp*(3256, T_1') = (y3, y4) = (33752, 33767) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
33514, T_1	N/A	*fp*(33514, T_1) = (y1, y2) = (348575, 348584) *fp*(33515, T_1) = (y3, y4) = (348585, 348598) **T** : ((y2 + 1) = y3)	*fp*(33514, T_1) = (y1, y2) = (348575, 348584) *fp*(33514, T_1') = (y3, y4) = (348576, 348586) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
40048, T_1	N/A	*fp*(40048, T_1) = (y1, y2) = (417419, 417436) *fp*(40049, T_1) = (y3, y4) = (417437, 417443) **T** : ((y2 + 1) = y3)	*fp*(40048, T_1) = (y1, y2) = (417419, 417436) *fp*(40048, T_1') = (y3, y4) = (417420, 417438) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))
50127, T_1	N/A	*fp*(50127, T_1) = (y1, y2) = (522843, 522846) *fp*(50128, T_1) = (y3, y4) = (522846, 522859) **F** : ((y2 + 1) = y3)	*fp*(50127, T_1) = (y1, y2) = (522843, 522846) *fp*(50127, T_1') = (y3, y4) = (522844, 522846) **F** : ((y3 = y1 + 1) ∧ (y4 = y2 + 1))

reliable. Metamorphic testing alleviates this issue through testing of metamorphic relationships and it augments the number of test cases by using random test values in its test data set.

Programs that can effectively make use of metamorphic testing must have strong metamorphic relationships. A strong metamorphic relationship is a relationship where it involves the execution of the core functionality and as such would effectively verify the function. A strong metamorphic relationship also has a high sensitivity to fault meaning that the relationship does not hold true for most input data. Both examples shown in this paper have strong metamorphic relationships. The likelihood for a metamorphic relation to detect errors increases when the metamorphic relation involves more invocations of the function. As demonstrated, some metamorphic relations are more sensitive to faults than others. This is due to the nature of the faults and how they are related to the metamorphic relation. Since it cannot be determined as to what errors could be revealed by metamorphic testing in advance, all available metamorphic relations, of which each one might have sensitivity to different faults, should be used as part of the test strategy.

One benefit of metamorphic testing lies in its ability to use random test values to augment the test data set to cover more of the test domain. Using random test values has the advantage of providing a larger test data set and this test set can be generated and used in testing through automation which allows testing to be done more efficiently. Another benefit of metamorphic testing is for a program to self-test without the need of test oracles, it provides a complementary approach to program testing. Like other testing approaches, metamorphic testing only demonstrates the presence but not the absence of faults. In other words, metamorphic testing does not prove the correctness of the program and so it should be used in addition to other testing methods such as special value testing.

In conclusion, the objective of this paper is to demonstrate that when special test values cannot sufficiently test a program, metamorphic testing could provide an effective way to complement the testing where test oracle is lacking. This has been achieved by using simple programs with strong metamorphic relationships. With that in mind, metamorphic testing and its ability to detect fault, perform program self-test and use randomly generated test data in different application domains could be researched into further.

Acknowledgement

References

[1] 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*. 1998

[2] T.Y. Chen, T.H. Tse and Z. Q. Zhou, "Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing", *Proceedings of the international symposium on Software testing and analysis,* 2002, pp.191-195.

[3] T.Y. Chen, T. H. Tse and Z.Q. Zhou, "Fault-based testing without the need of oracles", *Information and Software Technology*, vol. 45, no. 1, pp.1-9, 2003.

[4] S. L. Hantler, and J.C. King, "An introduction to proving the correctness of programs", *ACM Computing Surveys*,vol. 8, no. 3, pp.331-353, 1976.

[5] W.E.Howden, "Reliability of the path analysis testing strategy", *IEEE Transactions on Software Engineering,* vol. SE-2, no. 3, pp. 208-215, 1976.

[6] A. Mili, "*An introduction to Formal Program Verification*", New York: Van Nostrand Reinhold, 1985.

[7] E.J. Weyuker, "On testing non-testable programs", *The Computer Journal*, vol. 25, no. 4, pp.465-470, 1982.

[8] T. Wood, K. Miller and, R. Noonan "Local Exhaustive Testing: A Software Reliability Tool", *Proceedings of the 30th annual Southeast Regional Conference*, 1992, pp.77-84.