Research Journal of Applied Sciences, Engineering and Technology 6(16): 2951-2956, 2013
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: January 02, 2013 Accepted: February 08, 2013 Published: September 10, 2013

Research on Metamorphic Testing: A Case Study in Integer Bugs Detection

Yao Yi, Zheng Changyou, Huang Song and Ren Zhengping
College of Command Information System, PLA University of Science and Technology, China

Abstract: In order to solve Test Oracle problem which restricts the development of software testing techniques significantly, Metamorphic Testing is used to prove a way to determine the correctness of testing outputs with metamorphic relations between a series of outputs that correspond to a series of inputs in integer bugs detection, based on necessary properties of software under testing. A metamorphic relationship is proposed which can detect invisible integer bugs without oracle. It is shown in our case study that this method can detect some invisible errors which are difficult to be found in conventional approach and improve the efficiency of integer bugs detection.

Keywords: Integer bugs, metamorphic relationship, metamorphic testing, test oracle

INTRODUCTION

The basic process of software testing is in certain conditions, to provide test input and record output, then to compare actual output with expected output to determine whether the test passes (Heitmeyer, 2001). Software testing theory assumes that there must be a clear expected output, as a criterion to determine whether the test passes, however. But the reality is: software is not clear or difficult to obtain expected output and thus difficult to determine the correctness of the actual output. Therefore, there is a 'non-test program' which is determined by character of the test itself and not because of human factors (such as software development process lacks the necessary testing documentation, etc.). This question is so-called 'Test Oracle' problem in the software testing (Weyuker, 1982).

Test oracle is a mechanism for checking the correctness of testing results. An ideal test oracle could give "pass" / "no pass" judgments for test data. Unfortunately, an ideal oracle is either hard to obtain or difficult to apply. For instance, the oracle of numerical analysis software such as programs on partial differential equations is really hard to obtain. In encryption systems, where public key algorithm involves calculation of large integers, it is difficult to verify the correctness of the system. While testing Web search engines, it is also hard to judge whether the result is complete. When it comes to object-oriented software testing, the equivalence between two observed objects is nearly impossible to determine (Chen and Tse, 2001). In testing of GIS software, engineers are hard to judge the outputs of spatial distance calculation without test oracle.

However, even though the ideal oracle is hard to get sometimes, testing can still be carried out. By identifying the necessary relations among outputs of the target program and checking whether the actual outputs satisfy these relations, the correctness of the program would get enhanced to some degree and confidence to the correctness of programs could also be increased (Asrafi *et al*., 2011). Baresi and Young are the first to propose that necessary properties of program can be considered as an oracle. For example, a common method used for testing numerical programs is to check if the outputs satisfy certain properties, such as the property of $e^x \times e^{-x} = 1$ for exponential calculation programs. Such relations are widely used in testing of numerical programs. Program checkers and self-testing also make use of relations among outputs of object program to check its correctness by themselves. To design programs that check their work, one basic technique used is, first, determine the necessary properties of object program, then check if the program satisfies these properties with random inputs. The widely used properties are linear consistency and neighbor consistency. In software tolerance, the most relevant technique is data diversity that is re-describing the input in a different format and executing them again. This technique can lower the cost greatly compared with "N-version programming". Data diversity was first proposed for fault tolerance, not for fault detecting. Meanwhile, properties applied by data diversity are restricted by relations that can be recognized (Kuo *et al*., 2011a, b).

By expanding these techniques, Prof. (Chen *et al*., 1998) proposed a software testing technique based on Metamorphic Relations (MR), called Metamorphic Testing (MT), which checks the relation between inputs

Corresponding Author: Yao Yi, College of Command Information System, PLA University of Science and Technology, China, Tel.: 13851669990

and outputs to determine whether the program satisfies the necessary properties. As a result, it will be unnecessary to assume the existence an oracle for individual inputs and hence the test oracle problem can be alleviated. The necessary properties of the program under test are called metamorphic relations (Manolache and Kourie, 2001).

The integer bug is one of the main reasons that cause software calculation error. In computer program, the integer variables are expressed by fixed-bit-wide vector. When the value got by instruction operation is more than the value of storage capacity, integer overflow take places. When Europe launches rocket of Ariane5 firstly, because of an integer overflow in the procedure of a 64-bit floating point number into a 16-bit signed integer, the rocket control system instructing incorrectly resulted in disastrous consequences that the rocket vacated. In addition, because it can't afford to test every result of calculation, integer overflow in commercial software has not been detected by and large. For example, if an integer is disposed to be an unexpected value by a program and this unexpected value is then used for the array indexes or loop variable, it will produce software security vulnerabilities in the program.

This study presents a method to detect integer bugs by metamorphic relationship and validates this method by case study.

LITERATURE REVIEW

The test oracle problem has always been restricting the development of software testing. Researchers have been exploring the solutions to the problem in different applications. Then in Chen *et al*. (1998) and his fellows proposed MT, which is an effective method for test oracle problem. This technique boosts our confidence to the correctness of programs by checking whether it satisfies some necessary properties of the program, which are called MRs (Chen *et al*., 2009).

There are two understandings of the test oracle. Some people refer to the means of providing an expected outcome. Other people also include the process of comparing the actual outcome against the expected outcome. According these two explanations, we are going to give the definition of the test oracle.

Definition 1 (Test oracle in a narrow sense): Suppose T is the testing field, p (t) is the output of the program when given the input t, f (t) is the correct output of SUT where $t \in T$. Then test oracle is a mechanism for providing f (t) in a narrow sense.

Definition 2 (Test oracle in a wide sense): Suppose T is the testing field, p (t) is the output of the program when given the input t, f (t) is the correct output of SUT where $t \in T$. Then in a wide sense, test oracle is able to both provide f (t) and judge whether p (t) = f (t) is satisfied.

For example, program p_sin (x) is an implementation of sin (x), the test oracle in a narrow sense is all the sin (t) for $\forall t \in T$ and the test oracle in a wide sense refers to a mechanism that checks whether expression p_sin (x) = sin (x) is true for $t \in T$.

The test oracle used in this paper is referred to the one in a narrow sense without special statements.

Definition 3 (Metamorphic Relation (MR)): Suppose program P is the implementation of function f and x_1, x_2, …, x_n (n>1) is n groups of input for f, their corresponding outputs are f (x_1), f (x_2), …, f (x_n). If x_1, x_2, …, x_n satisfy relation r, it can be referred that f (x_1), f (x_2), …, f (x_n) satisfy relation r_f, that is:

$$r(x_1, x_2, \ldots, x_n) \Rightarrow r_f(f(x_1), f(x_2), \ldots, f(x_n)) \quad (1)$$

Then (r, r_f) is called a metamorphic relation of P.

Therefore, if P is correct, then it must satisfy the following comprehension:

$$R(I_1, I_2, \ldots, I_n) \Rightarrow r_f(P(I_1), P(I_2), \ldots, P(I_n)) \quad (2)$$

I_1, I_2, …, I_n are actual inputs of P corresponding with x_1, x_2, …,x_n and P (I_1), P (I_2), …, P (I_n) are the outputs. People could verify the correctness of P by checking whether expression (2-2) is satisfied while testing.

Suppose program P is correct, then the following expression should be satisfied: r (I_1, I_2, …, I_n) $\Rightarrow r_f$ (P (I_1), P (I_2), …, P (I_n)), where I_1, I_2, …,I_n is the input of P corresponding to x_1, x_2, …, x_n and P (I_1), P (I_2), …, P (I_n) is the output. We use x_1, x_2, …, x_n to represent the input in this paper. So if the outputs of test cases don't satisfy the above formula, then the hypothesis is wrong and there are faults in the program. Metamorphic relations are the key to judging the execution of the testing and their quality greatly affects the efficiency of testing. For different SUTs, there are usually more than one metamorphic relation, suppose MR_i (r_i, r_{fi}) is the i^{th} metamorphic relation of P and MR = {MR_1, MR_2,…} the set of metamorphic relations.

Based on the definition above, a formal definition of MR is proposed, which expresses the relation by symbolic logic as formula:

$$MR: \frac{P(x_i)=[P](x_i)}{r(x_1,x_2,..,x_n) \rightarrow r_f([P](x_1),[P](x_2),...,[P](x_n))} \quad (3)$$
$$(i=1,2,...,n)$$

Then, the relation r is called as Input Relation of Metamorphic Relation (MR_IR), which denote as IR = {(x_1, x_2, …, x_n) |r (x_1, x_2, …, x_n)}. The relation r_f is called as Output Relation (MR_OR), which also denote as OR = {([P] (x_1), [P] (x_2), …, [P] (x_n)) |r_f ([P]

(x_1), [P] (x_2), …, [P] $(x_n))\}$. And the relation [P] (x) is called as Self Relation of Metamorphic Relation. Self Relation Therefore, metamorphic relation could be repressed as MR = [IR, SF, OR].

Definition 4 (Original test cases): It is also called Original Test Input recorded as OTI. Suppose there is a metamorphic relations (r, r_f) and its input is r $(x_1, x_2, \ldots, x_n)$, then the OTI is test cases from r $(x_1, x_2, \ldots, x_n)$ which are generated with other testing methods, such as special testing, random testing and iterative testing.

Definition 5 (Follow-up test cases): It is also called Follow-up Input (FTI). Suppose we have a metamorphic relation (r, r_f) and its input is r $(x_1, x_2, \ldots, x_n)$. FTI is all the test cases from r $(x_1, x_2, \ldots, x_n)$ except original test cases. Follow-up test cases are generated based on metamorphic relations.

Suppose the input of a MR is r $(x_1, x_2, \ldots, x_n)$, then it can be recorded as r (OTI, FTI). For example, for program p_sin (x), there is a MR_{sin1}: $\sin^2(x_1) + \sin^2(x_2) = 1$, where $x_2 = \pi/2 - x_1$, x_1 is the original test case and x_2 is the follow-up test case.

METHODS OF INTEGER BUGS DETECTION

Integer bugs detection currently contains three methods. It consists of prior condition, error detection and back conditions. Among them, the prior condition is used to check whether an error occurs before perform operation. For example, a prior condition of division by zero is that the divisor is 0. These prior conditions can often abstracted to be formalization rules in advance and then use static code analysis tool based on these rules to review codes. Error detection required to determine whether the errors occur in the process of implementation. Because integer errors are caused by limit of machine, the operating system and compiler can provide a handling mechanism to find positive overflow error. Back conditions is to carry out the operations firstly and then to get the conclusion by comparing the actual result with the expected result. Back condition is also one of the most common detection methods.

Three methods of Prior condition, error detection and error condition have advantages and disadvantages of themselves, but they all are not suitable for integer bugs if the expected results can not directly be got and the program output which is normal is not the correct. Under this circumstance, testers are difficult to get efficient and accurate test results by use of these three methods generally.

Methods of integer bugs detection based on metamorphic testing: For the 'Test Oracle' problem in the method of the integer bugs detection, method of the integer bugs detection based on metamorphic test is proposed, which is essentially a detection method based on validation of correctness. If an integer error occurs, the program must calculate to get an inaccurate value and then if this inaccurate value is used in the next step calculation, the program will eventually either collapse, or calculate to get an unexpected output. That is, when the inputs of software meet some certain characters, the corresponding output of software will also meet the corresponding character.

Before the specific description of this method, formal definitions of the concepts which are required in the method are given.

Definition 1: It is assumed that program P is an implementation of the function f. $x_1, x_2, \ldots, x_n$, (n>1) are n-group variables for function f and f (x_1), f (x_2), …, f (x_n) are corresponding outputs for function f. If $x_1, x_2, \ldots, x_n$ satisfy the relation r among themselves and f (x_1), f (x_2), ..., f (x_n) satisfy the relation rf:, (r, rf) is recognized as the metamorphic relationship of program P .

Definition 2: For the same program P, Metamorphic relationships (r, rf) which need to verify or to extract always are not only one. It is shown that $R_i = (r_i, r_{fi})$ denotes the i-th metamorphic relationship of the program P and that S (R) = $\{R_1, R_2, \ldots\}$ denotes the set of metamorphic relationships of the program P.

Method of the integer bugs detection based on metamorphic relationship includes three steps:

Step 1: Select the source test cases. For program P, I_1, I_2, …, I_n are selected as inputs corresponding to $x_1, x_2, \ldots, x_n$ in program P. That is, source test case (I_1, I_2, …, I_n) are gained.

Step 2: Select the correct metamorphic relationship to generate follow-up test cases. We choose the appropriate metamorphic relation R = (r, rf) of program P. It is assumed that the program P is correct and then r $(I_1, I_2, \ldots, I_n) =>_{r_f}$ (P (I_1), P (I_2), …, P (I_n)) is generated from definition 1. It means that follow-up test cases to be derived from the test case. If there are a variety of metamorphic relationships, can also choose a number of metamorphic relations, some metamorphic relationships $R_1, R_2, \ldots, R_n$ can be selected to build many follow-up test cases in order to enhance the veracity of test.

Step 3: Compared results from source test cases with Results from follow-up test cases in order to judge whether metamorphic relationship is obeyed. If program P is correct, P obeys r $(I_1, I_2, \ldots, I_n) =>_{r_f}$ (P (I_1), P (I_2), …, P (I_n)) in which (P (I_1), P (I_2), …, P (I_n) are the corresponding output. So if the test case which is running does not meet the formula above, the assumption does not correct and it means the program has errors.

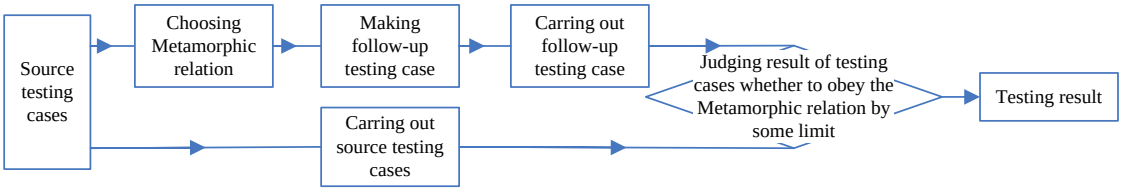

Fig. 1: Process of integer bugs detection based on metamorphic testing

Figure 1 shows process of integer bugs detection based on metamorphic testing.

CASE STUDY

Programs for calculating graphics area on Cartesian coordinate have been applied to computer graphics, physics and other fields. Here, calculation of graphics area on Cartesian coordinate is used as case studies for research on method of integer bugs detection based on metamorphic testing.

The program can be described as: vertex coordinates of convex polygon are input in the txt file; the program reads the txt file as input and then calculates area and perimeter of convex polygon. The strategy of area calculation is to divide convex polygon into several triangles and then each triangle is calculated according to Heron. Finally, the sum area is calculated.

Variation design: The statement "S = (s1 + s2 + s3) /2" is modified as "S = (s1 + s2 + (int) s3) /2", while the program of variation version which contains integer errors is recorded as mutant1 (if variable s3 is changed from double type into integer type, truncation errors may be appeared). Codes of program are shown as below:

```
/* calculate triangle area with Heron's formula */
double area (triangle a)
{
double s1, s2, s3, S, area;
s1 = side (a.v1, a.v2);
s2 = side (a.v2, a.v3);
s3 = side (a.v3, a.v1);
S = (s1 + s2 + (int) s3) /2; //Correct version: S = (s1 + s2 + s3) /2;
area = sqrt (S* (S - s1) * (S - s2) * (S - s3));
return area;
}
```

Metamorphic relationship: By using similar triangles character that area and perimeter are proportional to side length, the following metamorphic relationships can be gained, as shown in Fig. 2.

The metamorphic relationships are separately tagged as MR1 (ratio of area) and MR2 (the ratio of perimeter), then some equations exist as below:

$$triangle(v_1, v_2, v_3), triangle(v_1, v_2', v_3'), triangle(v_1, v_2'', v_3''), triangle(v_1, v_2''', v_3''')$$

$$\text{and } \overrightarrow{v_1v_2} = 2\overrightarrow{v_1v_2'} = 4\overrightarrow{v_1v_2''} = 8\overrightarrow{v_1v_2'''}\ \overrightarrow{v_1v_3} = 2\overrightarrow{v_1v_3'} = 4\overrightarrow{v_1v_3''} = 8\overrightarrow{v_1v_3'''}$$

$$\Rightarrow \begin{cases} \dfrac{perimeter(v_1, v_2, v_3)}{perimeter(v_1, v_2', v_3')} = \dfrac{perimeter(v_1, v_2', v_3')}{perimeter(v_1, v_2'', v_3'')} = \dfrac{perimeter(v_1, v_2'', v_3'')}{perimeter(v_1, v_2''', v_3''')} = 2 \text{(ratio of perimeter)} \\ \dfrac{area(v_1, v_2, v_3)}{area(v_1, v_2', v_3')} = \dfrac{area(v_1, v_2', v_3')}{area(v_1, v_2'', v_3'')} = \dfrac{area(v_1, v_2'', v_3'')}{area(v_1, v_2''', v_3''')} = 4 \text{(ratio of area} \end{cases}$$

Testing cases design: The integer errors usually occur near the border of input field, so two sets of data are randomly generated in the two fields of [64535, 66535] and [-66535, -64535] and the two sets of data were randomly combined into 10 groups of the source test input, as shown in Table 1.

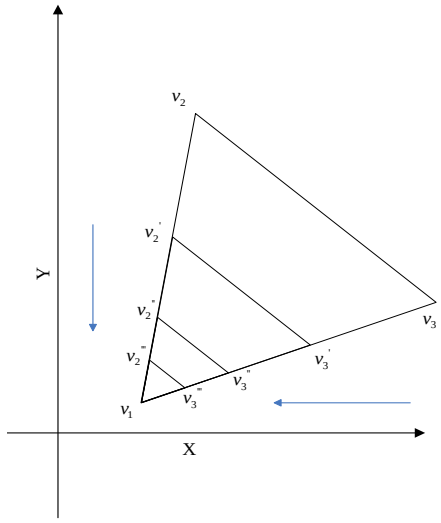

Fig. 2: Area and perimeter are proportional to side length in similar triangles

In accordance with MR1 and MR2, follow-up test inputs are generated as below:

$$\frac{v_1+v_2}{2}=v_2',\ \frac{v_1+v_3}{2}=v_3';\ \ \frac{v_1+v_2'}{2}=v_2'',\ \frac{v_1+v_3'}{2}=v_3'';$$
$$\frac{v_1+v_2''}{2}=v_2''',\ \frac{v_1+v_3''}{2}=v_3''''$$

Because of errors of floating point calculations, two formulas that MR1 (circumference ratio) and MR2 (area ratio) should be converted in order to eliminate the impact of these errors. The new formulas are as follows:

$$\begin{cases}\left|\dfrac{perimeter(Si)}{perimeter(Y_1i)}-2\right|=\sigma_1,\left|\dfrac{perimeter(Y_1i)}{perimeter(Y_2i)}-2\right|=\sigma_2,\left|\dfrac{perimeter(Y_2i)}{perimeter(Y_3i)}-2\right|=\sigma_3\\ \text{and}\ \left|\dfrac{area(Si)}{area(Y_1i)}-4\right|=\varepsilon_1,\left|\dfrac{area(Y_1i)}{area(Y_2i)}-4\right|=\varepsilon_2,\left|\dfrac{area(Y_2i)}{area(Y_3i)}-4\right|=\varepsilon_3\\ i\in[1,10],\ \Delta\overline{s}_1=\dfrac{1}{3}\sum_{j=1}^{3}\sigma_j<10^{-3},\Delta\overline{s}_2=\dfrac{1}{3}\sum_{j=1}^{3}\varepsilon_j<10^{-3}\end{cases}$$

Y_1i, Y_2i, Y_3i are generated as three sets of follow-up test inputs by S_i in accordance to MR1 and MR2. If the conditions are obeyed, we call test cases (S_i, Y_1i, Y_2i, Y_3i) satisfy MR1 and MR2 of metamorphic relationships.

TESTING RESULTS

Through running the source version and follow-up version of the program with gcc compiler under the circumstance of Linux, results are gained as shown in Table 2. Compared to test results executed in versions before and after variation of the program, we can conclude that test cases set {(S_i, Y_1i, Y_2i, Y_3i)| i = 1, 2, 3,..., 9, 10} generated by MR1 and MR2 can detect

Table 1: Source test input

	v_1		v_2		v_3	
Input (no.)	x1	y1	x2	y2	x3	y3
S1	65966.1	66370.0	65977.6	66090.3	65757.0	65819.9
S2	66395.3	65859.1	65878.5	65961.3	66281.4	66247.0
S3	-64788.6	65607.0	66361.0	65831.7	65741.6	-65139.1
S4	-65328.4	-666251.6	66155.9	-65364.1	-65463.0	-64683.6
S5	-65032.0	66382.5	66263.8	-64674.7	-64818.4	65959.3
S6	-64998.2	-65032.9	-65250.1	65727.8	66038.0	66371.1
S7	65921.9	65959.3	-64823.0	65937.1	66071.8	65997.4
S8	65676.0	66371.1	-65110.7	-64767.3	-64700.0	-65297.6
S9	-65313.0	65997.4	-64698.9	-65148.1	-65210.9	-65032.9
S10	-65364.1	-65297.6	-65072.6	-65394.0	-64687.5	-64616.0

Table 2: Test results of source version and mutant1 by MR1 and MR2

Test Case	Source $\Delta\overline{s_1},\Delta\overline{s_2}$	Mutant 1 $\Delta\overline{s_1},\Delta\overline{s_2}$
($S1, Y_1 1, Y_2 1, Y_3 1$)	$\Delta\overline{s_1}=2.39013\times10^{-4},\Delta\overline{s_2}=7.11892\times10^{-5}$	$\Delta\overline{s_1}=8.49250\times10^{-2},\Delta\overline{s_2}=7.11892\times10^{-5}$
($S2, Y_1 2, Y_2 2, Y_3 2$)	$\Delta\overline{s_1}=1.49831\times10^{-4},\Delta\overline{s_2}=1.87129\times10^{-5}$	$\Delta\overline{s_1}=1.98616\times10^{-2},\Delta\overline{s_2}=1.87129\times10^{-5}$
($S3, Y_1 3, Y_2 3, Y_3 3$)	$\Delta\overline{s_1}=1.69067\times10^{-9},\Delta\overline{s_2}=5.96740\times10^{-8}$	$\Delta\overline{s_1}=1.03223\times10^{-4},\Delta\overline{s_2}=5.96740\times10^{-8}$
($S4, Y_1 4, Y_2 4, Y_3 4$)	$\Delta\overline{s_1}=1.01271\times10^{-6},\Delta\overline{s_2}=5.43897\times10^{-8}$	$\Delta\overline{s_1}=1.01261\times10^{-6},\Delta\overline{s_2}=5.43897\times10^{-8}$
($S5, Y_1 5, Y_2 5, Y_3 5$)	$\Delta\overline{s_1}=8.58888\times10^{-7},\Delta\overline{s_2}=1.43737\times10^{-7}$	$\Delta\overline{s_1}=6.10469\times10^{-2},\Delta\overline{s_2}=1.43737\times10^{-7}$
($S6, Y_1 6, Y_2 6, Y_3 6$)	$\Delta\overline{s_1}=1.46789\times10^{-9},\Delta\overline{s_2}=7.44673\times10^{-8}$	$\Delta\overline{s_1}=8.64704\times10^{-5}\ \ \Delta\overline{s_2}=7.44673\times10^{-8}$
($S7, Y_1 7, Y_2 7, Y_3 7$)	$\Delta\overline{s_1}=3.09947\times10^{-6}\ \ \Delta\overline{s_2}=5.09306\times10^{-8}$	$\Delta\overline{s_1}=5.47477\times10^{-1}\ \ \Delta\overline{s_2}=5.09306\times10^{-8}$
($S8, Y_1 8, Y_2 8, Y_3 8$)	$\Delta\overline{s_1}=1.70404\times10^{-4},\Delta\overline{s_2}=2.87375\times10^{-7}$	$\Delta\overline{s_1}=1.42761\times10^{-2},\Delta\overline{s_2}=2.87375\times10^{-7}$
($S9, Y_1 9, Y_2 9, Y_3 9$)	$\Delta\overline{s_1}=5.72487\times10^{-7},\Delta\overline{s_2}=2.03018\times10^{-7}$	$\Delta\overline{s_1}=1.62891\times10^{-2},\Delta\overline{s_2}=2.03018\times10^{-7}$
($S10, Y_1 10, Y_2 10, Y_3 10$)	$\Delta\overline{s_1}=1.88750\times10^{-4},\Delta\overline{s_2}=1.60825\times10^{-5}$	$\Delta\overline{s_1}=3.42968\times10^{-4},\Delta\overline{s_2}=1.60825\times10^{-5}$

abnormal outputs caused by variation, i.e., mutant 1 flawed. Therefore, it's proved that method of integer bugs detection based on metamorphic relationship is validated to find hidden integer bugs.

It is shown that defects of mutant 1 are detected only by the MR1. The reason is that MR2 which is metamorphic relationship obtained by the perimeter of the triangle and the mutant 1's function is to calculate area of a triangle, so it is unable to detect defects of mutant 1 by the use of MR2.

CONCLUSION AND RECOMMENDATIONS

In this study, method of integer bugs detection based on metamorphic relationship is proposed. It is proved by case studies that this method of metamorphic relationship can detect the hidden unexpected failure which traditional testing techniques can't detect.

Because of certain blindness of choosing source test input and metamorphic relationship, optimization algorithm of test case generation and selection of metamorphic relationship are the research direction for the future. At present, research method by use of the evolutionary algorithms combined with metamorphic relationship are proposed and error detection efficiency of test cases is improved (Dong *et al*., 2010), but whether this method is effective for the integer bugs detection need to be further studied.

ACKNOWLEDGMENT

This study is supported by National High Technology Research and Development Program of China (No: 2009AA01Z402), China Postdoctoral Science Foundation (No: 20110491843) and the Natural Science Foundation of Jiangsu Province, China (Grant No. BK2012059, BK2012060). Resources of the PLA Software Test and Evaluation Centre for Military Training are used in this research.

REFERENCES

Asrafi, M., H. Liu and F.C. Kuo, 2011. On testing effectiveness of metamorphic relations: A case study. Proceeding of 5th International Conference on Secure Software Integration and Reliability Improvement. IEEE Computer Society Washington, DC, USA, pp: 147-156.

Chen, H.Y. and T.H. Tse, 2001. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. ACM T. Softw. Eng. Meth., 10: 56-109.

Chen, T.Y., S.C. Cheung and S.M. Yiu, 1998. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01.

Chen, T.Y., J.W.K. Ho, H. Liu and X. Xie, 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. BMC Bioinformatics, 10: 24-35.

Dong, G.W., S.Z. Wu, G.S. Wang, T. Guo and Y.G. Huang, 2010. Security assurance with metamorphic testing and genetic algorithm. Proceeding of IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, pp: 368-373.

Heitmeyer, C.L., 2001. Applying practical formal methods to the specification and analysis of security properties. Proceeding of the International Workshop on Information Assurance in Computer Networks: Methods, Models and Architectures for Network Computer Security (MMM ACNS 2001) Springer-Verlag, Heidelberg, St. Petersburg, Russia, LCNS 2052: 84-89.

Kuo, F.C., S. Liu and T.Y. Chen, 2011a. Testing a binary space partitioning algorithm with metamorphic testing. Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11) ACM, New York, USA, pp: 1482-1489.

Kuo, F.C., T.Y. Chen and W.K. Tam, 2011b. Testing embedded software by metamorphic testing: A wireless metering system case study. Proceeding of 2011 IEEE 36th Conference on Local Computer Networks. Bonn, pp: 291-294.

Manolache, L.I. and D.G. Kourie, 2001. Software testing using model programs. Software Pract. Exper., 31: 12 11-1236.

Weyuker, E.J., 1982. On testing non-testable programs. Comput. J., 25: 465-470.