IJCSC International Journal of Computer Science & Communication Vol. 1, No. 2, July-December 2010, pp. 79-82

Designing Robust Test Case using Mutation Testing

Ajay jangra[1], Ruchi Pahuja[2], Chander Diwakar[3] & Priyanka[4]

[1,2]CSE Deptt. U.I.E.T., Kurukshetra University, Kurukshetra
[3]ECE Deptt. Kurukshetra Institute of Engineering & Technology, Kurukshetra
Email: [1]er_jangra@yahoo.co.in, [2]ruchi_ndatasystem@yahoo.com, [3]chander_cd@rediffmail.com, [4]priyanka.jangra@gmail.com

ABSTRACT

Testing is essential tool for good quality software. It examines the functionality and performance behavior of a program by testing it with an intention to finding errors/bugs. Mutation testing is based on change and check testing strategy. Mutation testing may also be used to check the effectiveness of a test case. In this paper, we implement mutation testing strategy to check the effectiveness of automated test case. For this, we generate 4 programs namely (i) sample.c – the original program for adding two numbers using files in C, (ii) genmutant.c – the program which generate mutants by changing the arithmetic operator in original program, (iii) chkmutant.c – the program which will then compare the output of original program with the output of mutated program and finally (iv) resmutant.c – which will display the output in graph form.

Keywords: Mutation Testing, Automated Test Case, Mutant, Software Testing, Killed Mutent.

1. INTRODUCTION

Software testing describes the relationship of an input with respect to outputs of any software. Testing process used to ensure that the program realizes the function and check the performance of software. Testing process generally execute with an intention to find errors in a given program. Functional and Structural testing are two categories of software testing, where functional testing (Black-Box Testing) focus on the functional behavior of a software without checking its internal structure and structural testing (White-Box Testing) examines the internal structure of the program in detail. Validating the software using testing is a cost-intensive process. Generally testing includes consumption of about half of the total cost of software development and maintenance. It is not uncommon for a software organization to spend 40% of its effort on testing. The objective of functional and structural testing is same but they are often conducted separately because of lack of techniques and tools to integrate both testing strategies. In Software Development Life Cycle (SDLC) level based testing techniques are used like Unit testing – which tests the individual module of software, Integration testing – check the structure of two or more integrated components, System testing – examines the end to end quality of the complete software and Acceptance testing is performed when the system is handed over to the customers. [7, 8, 11, 13]

Why to Perform Testing?

Although software testing is itself an expensive and time consuming activity, but we can't ignore its importance. Testing builds the confidence of developers as well as users about quality and performance of software. It also optimizes the functionality of any software by detecting errors and faults etc. It is not feasible (economically feasible) to test the software for all possible combinations of input sets. In consideration with costs, efforts and time factors of testing, a minimum testing threshold should be achieved where all inputs must executes at least once and tests for highly possible errors/bugs.

Test Case Generation

Test case consists of set of inputs and a list of expected outputs. Inputs are basically of two types: pre-conditions (conditions that occur prior to the execution of test case) and the actual inputs. Expected output is also of two types: post-conditions and actual outputs. Each test case is associated with an identity. A good test case has higher capability of finding the errors and a test suite is a collection of set of test cases. Earlier testing process performed manually which seemed as very slow, high cost, time consuming and less accurate practice. Automated test case generation techniques can be used to simplify this process. Here, test cases are generated and executed automatically with an intention to find errors and check the functionality of software. Mutation testing behaves as a good tool for automated testing [10].

Mutation Testing

Mutation testing performs "change and check" testing strategy. Original program is slightly modified and then executed. The output of original program and that modified program with respect to the same input set are then compared. For example - we have a program P and slightly modified (mutated) program P' and Let I be the input set. With execution of same input set I, program P gives output O and program P' give output O'.

$I \rightarrow P \rightarrow \rightarrow O$ $I \rightarrow P \rightarrow \rightarrow O'$

If ($O' \neq O$) then it means, our test case is adequate and the functionality of the program is good. Otherwise, if ($O' = O$), then our test case is inadequate and the functionality of the program is poor. Consider the example for mutant generation as shown in Fig.1.

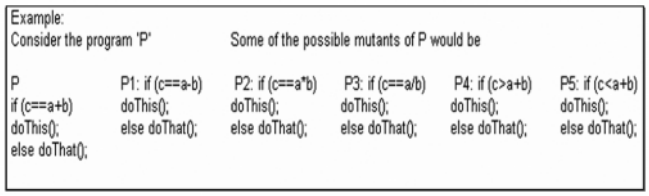

Fig. 1: Mutant Generation

If the value of $a = 2$ and $b = 2$ then P2 is an equivalent mutant of P because it is not possible to find a test case that can ever kill this mutant. [10]

Literature Survey

The approach to seeding some faults in programs through mutation operators was posed originally by DeMilllo et al.[2].Mutation testing is a fault based testing technique for unit level testing of software. Like other fault based testing technique, the main purpose of mutation testing is to measure the quality of test input set. However, it can also be used to reduce the size of test set, to generate effective test data and to compare techniques for verification. [3]

Mutation testing is also used to increase the confidence that a program P (or a specification) is correct by producing through small syntactic changes, a set of mutant elements that are similar to product under test. These changes are based on an operator set called mutation operators. A test case set that is capable of causing behavioral differences between P (or S) and each one of its mutants is then generated. Very simple operators are usually defined based on the competent programmer hypothesis, which affirms that a program produced by a competent programmer is either correct or near correct. Another hypothesis considered by Mutation testing is the coupling effect that, according to DeMillo et al [14] can be described as "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it would also implicitly distinguish more complex errors. Mutation testing consists of four steps: mutant generation: execution of the program P (or specification) based on defined test set T: mutant execution; and adequacy analysis. It is recognized that the major obstacle to the use of the Mutation Testing is its computational cost due to the high no. of mutants and to the need to analyze them aiming to determine the equivalent ones. [12, 14]

In [1] Garrett Kaminski and Paul Ammann state that Mutation testing is a technique for generating high quality test data and to improve the efficiency of test data. They also state that logic mutation testing is currently inefficient for three reasons:

- Logic mutation produces multiple identical mutants i.e. same mutant is generated multiple times.
- Mutants are generated that are guaranteed to be killed by inputs that kill some other generated mutant.
- Current mutation tools lack logic mutation operators that generate mutants which, when killed, guarantee killing of many other mutants.

These inefficiencies cause excess mutants to be generated and reduce fault detection capability. This paper presents a technique to reduce mutant set size while increasing fault detection, assuming that predicates are in minimal DNF. This paper improves logic mutation testing by 1) Extending a logic fault hierarchy to include current mutation operators. 2) Introducing new logic mutation operators based on existing faults in the hierarchy, 3) introducing new logic mutation operators having no corresponding faults in the hierarchy and extending the hierarchy to include them, and 4) addressing the precise effects of equivalent mutants on the fault hierarchy.

Proposed Work

Effectiveness means how good (or efficient) is the test case in finding an errors and the main objective of this paper is to check the effectiveness of test cases and to kill mutants as much as possible. For checking the effectiveness of test cases using mutation testing, we generate mutated program by changing the arithmetic operators (+,–, * and /) in the original program. To do this, we generate 3 programs in C. The first original program (1_sample.c) will add 2 number ('a', 'b') and write the value of their sum in variable 'c'. Second program (2_genmutant.c) will generate mutants (M1, M2, M3 and M4 versions) of the original program. For generating mutants, this program will read each and every character of the original program (1_sample.c) until it finds the location of plus (+) operator in that program.

After finding the location, it will assign 43 (*ASCII value of "+" operator*) at that location and then write it into "mutant1.c" i.e. generate mutant1. Similarly, again at that location, it will assign 45 (*ASCII value of "-" operator*) at that location and then write it into "mutant2.c" i.e. generate mutant2. Again, it will assign 42 (*ASCII value of "*" operator*) at that location and then write it into "mutant3.c" i.e. generate mutant3. Then, it will assign 47 (*ASCII value of "/" operator*) at that location and then write it into "mutant4.c" i.e. generate mutant4.

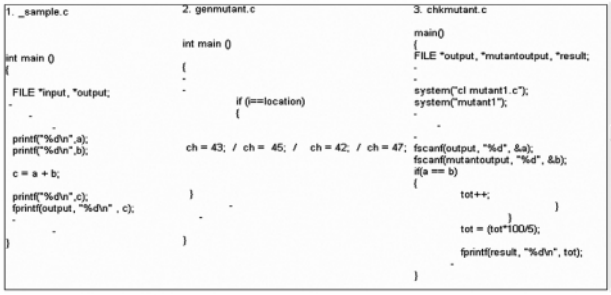

Fig. 2: Programs 1_sample.c, 2_genmutant.c and 3_chkmutant.c

Next program (3_chkmutant.c) will calculate the % age of survival of mutants. For this, program will check the output of original program with output of mutants (M1, M2, M3 and M4 versions). If output of the original program and the mutant is same, it means our test case is inadequate and Survival of mutants is more. So, In this case, mutant is called alive. If the mutant and the original program generate different outputs for a test case, then our test case is adequate and the mutant is called killed mutant.

2. IMPLEMENTATION

For implementing the concept of Mutation Testing, we randomly take 5 input set and each input set contains 5 paired inputs. As our original program contains the sum of these 5 paired inputs. So, now we check the original output with the output of each mutant. As shown in table,

Input test case-I contains 5 paired inputs	(7, 3),	(2, 2),	(9, 4),	(10,3),	(16,20)
Input test Case-II contains 5 paired inputs	(5,8),	(9,2),	(12,4),	(2,2),	(2,2)
Input test Case-III contains 5 paired inputs	(9,3),	(2,6),	(20,5),	(18,1),	(6,6)
Input test Case-IV contains 5 paired inputs	(17,6),	(2,2),	(25,9),	(2,2),	(2,2)
Input test Case-V contains 5 paired inputs	(2,2),	(2,2),	(18,6),	(2,2),	(2,2)

Table 1
Test Case v/s Mutant Versions

	Inputs	*Original o/p*	*Mutant-M¹*	*Mutant-M²*	*Mutant-M³*	*Mutant-M⁴*
Test case-I	(7,3) (2,2) (9,4) (10,3) (16,20)	10,4,13,13,36	10,4,13,13,36	4,0,5,7,-4	21,4,36,30,320	2,1,2,3,0
Test case-II	(5,8),(9,2),(12,4),(2,2),(2,2)	13,11,16,4,4	13,11,16,4,4	-3,7,8,0,0	40,18,48,4,4	0,4,3,1,1
Test case-III	(9,3),(2,6),(20,5),(18,1),(6,6)	12,8,25,19,12	12,8,25,19,12	6,-4,15,17,0	27,12,100,18,36	3,0,4,18,1
Test case-IV	(17,6),(2,2),(25,9),(2,2),(2,2)	23,4,34,4,4	23,4,34,4,4	11,0,16,0,0	102,4,225,4,4	2,1,2,1,1
Test case-V	(2,2),(2,2),(18,6),(2,2),(2,2)	4,4,24,4,4	4,4,24,4,4	0,0,12,0,0	4,4,108,4,4	1,1,3,1,1

As shown in the above given table:

Case – 1: The original output and output of mutant (M1) is same. So Survival of mutant is 100%.

Case – 2: The original output and output of mutant (M2) is entirely different. So Survival of mutant is 0%.

Case – 3: Again check the original output with output of mutant (M3). There is one similar entity at 2nd position in both the outputs. So, in this case we will calculate the %age of survival of mutants in the form of:

$$\frac{\text{Number of matching entities}}{\text{Total number of entities}} * 100 \text{ i.e. } \frac{1}{5} * 100 = 20\%$$

Case – 4: Again the original output and output of mutant (M4) is entirely different. So Survival of mutant is 0%. Finally, the result will display in the graph form.

As shown in the *table-2* for all the 5 input sets (I^1, I^2, I^3, I^4 and I^5), the output of mutant (M1) and original output are found equal because both adopt the same operator (i.e. +operator).So, in this case survival of mutants is 100% (i.e. 0% mutants killed).

In M2 and M4, mutant survival is 0% because not a single output is matched with original output and 100% mutants are killed.

Table 2
Applied Input Set v/s Survival of mutant

	Survival of mutant			
Applied Input Set	*Test Case 1*	*Test Case 2*	*Test Case 3*	*Test Case 4*
I^1 = (7,3),(2,2),(9,4),(10,3),16,20)	100%	0%	20%	0%
I^2 = (5,8),9,2),(12,4),(2,2),(2,2)	100%	0%	40%	0%
I^3 = (9,3),(2,6),(20,5),(18,1),(6,6)	100%	0%	0%	0%
I^4 = (17,6),(2,2),(25,9),(2,2),(2,2)	100%	0%	60%	0%
I^5 = (2,2),(2,2),(18,6),(2,2),(2,2)	100%	0%	80%	0%

But in case of M3, for input set (I1) – survival of mutant is 20% and for input set (I2) – survival of mutant is 40% and for input set (I3) – survival of mutant is 0% and for input set (I4) – survival of mutant is 60% and for input set (I5) – survival of mutant is 80%.

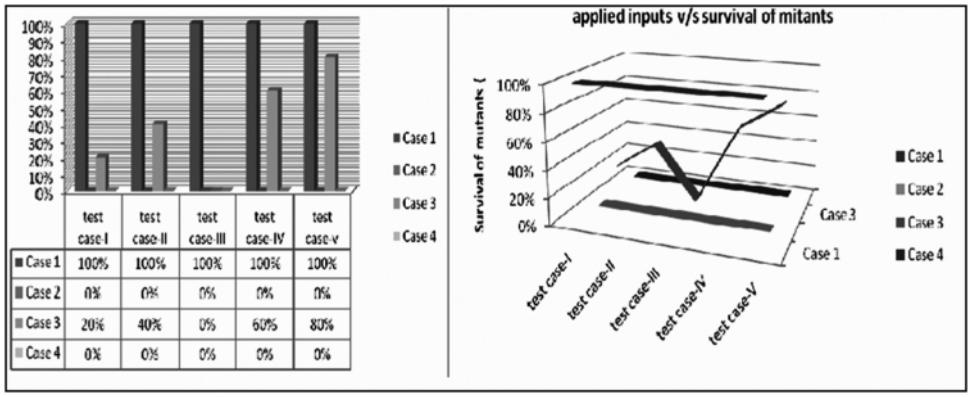

Fig.3: Graph and Survival of Mutants for all Input Test Cases

3. CONCLUSION

In this paper, we examine the utilities of mutation testing to check the effectiveness of given automated test cases by creating mutants (changing arithmetic operators of a given program). We execute four different programs (sample.c, genmutant.c, chkmutant.c, resmutant.c) to calculate the survival of mutants for all five input sets. We found output of mutant 1, mutant 2 and mutant 4 behave in the same manner for all input sets but output of mutant 3 behave differently w.r.t. supply input set.

REFERENCES

[1] Kaminski, G.; Ammann, P., "Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing," *Software Testing Verification and Validation, 2009*. ICST '09. International Conference on, vol., no., pp.386-395, 1-4, April 2009.

[2] Chixiang Zhou; Frankl, P., "Mutation Testing for Java Database Applications," *Software Testing Verification and Validation*, 2009. ICST '09. International Conference on, vol., no., pp.396-405, 1-4, April 2009.

[3] Yue Jia; Harman, M., "Constructing Subtle Faults Using Higher Order Mutation Testing," *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, vol., no., pp.249-258, 28-29 Sept. 2008.

[4] Shufang Lee; Xiaoying Bai; Yinong Chen, "Automatic Mutation Testing and Simulation on OWL-S Specified Web Services," *Simulation Symposium*, 2008. ANSS 2008. 41st Annual, vol., no., pp.149-156, 13-16 April 2008.

[5] Chuanming Jing; Zhiliang Wang; Xingang Shi; Xia Yin; Jianping Wu, "Mutation Testing of Protocol Messages Based on Extended TTCN-3," *Advanced Information Networking and Applications*, 2008. AINA 2008. 22nd International Conference on, vol., no., pp.667-674, 25-28 March 2008.

[6] Ferrari, F.C.; Maldonado, J.C.; Rashid, A., "Mutation Testing for Aspect-Oriented Programs," *Software Testing, Verification, and Validation*, 2008 1st International Conference on, vol., no., pp.52-61, 9-11 April 2008.

[7] F Tao ; Kasturi Bidarkar, "A Survey of Software Testing Methodology".

[8] Sami Beydeda; Volker Gruhn; Michael Stachorski, "A Graphical Class Representation for Integrated Black- and White-Box Testing".

[9] URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4144731&isnumber=4144719.

[10] Masud, M.; Nayak, A.; Zaman, M.; Bansal, N., "Strategy for Mutation Testing using Genetic Algorithms," *Electrical and Computer Engineering*, 2005. Canadian Conference on, vol., no., pp.1049-1052, 1-4 May 2005.

[11] Frank Lammermann ; Joachim Wegener, "White Box Testing by Means of Software Measures", The Sixth Metaheuristics International Conference 2005.

[12] Fabbri, S.C.P.F.; Maldonado, J.C.; Sugeta, T.; Masiero, P.C., "Mutation Testing Applied to Validate Specifications based on Statecharts," *Software Reliability Engineering*, 1999. Proceedings. 10th International Symposium on, vol., no., pp.210-219, 1999.

[13] Danhua Shao ;Sarfraz Khurshid; Dewayne E. Perry, "A Case for White-box Testing Using Declarative Specifications Poster Abstract" *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2007.

[14] DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer*, Vol. 11(4), pp. 34-41, 1978.