A New Approach for Data Mutation Based Test Case Generation

Dr.Y. Kalpana and Pramod Kumar Mallick

Abstract--- Mutation testing is taken as a very powerful tool dependent testing technique but it is too costly. It is a pleasant way of using large number of test requirements for ensuring quality. On the other hand it also need of heavy automation. It is considered costly because of the high number of requirements it creates compared to other testing techniques and it is tool dependent for the same reason. In this paper, we present an incipient technique called data mutation predicated on engendering an immensely colossal number of test data from initial test cases either manually or with some automatic test case generation method. It is influenced by some mutation testing methods, but varies from the manner that mutation operators are defined and utilized. While mutation testing is a technique for quantifying test adequacy, data mutation is a technique of test case generation. In traditional mutation testing, mutation operators are acclimated to convert the program under test. In contrast, mutation operators in our task are applied on input data to engender test cases, hence called data mutation operators. In this paper will be implemented with the approach on testing an automated modelling implement to describe the applicability of the proposed method.

Keywords--- Mutation Testing, Test Cases, Mutation Operator

I. Introduction

Software testing is a serious and very important activity for assessing and achieving the quality of a software product [1]. The goal is to identify potential faults in order to increase the quality of software products. However, in general, it is not possible to fully automate software testing activities because of constraints related to undecidable problems. Mutation Testing is a method of inserting some faults into programs to test whether the tests pick them up, thereby validating or invalidating the tests. Mutation testing is concerned as a testing criterion in effective manner. The main goal of mutation testing is as follows: (i) to evaluate the quality of the tests by performing them on mutated code (ii) to utilize these evaluation to avail and construct more adequate tests (iii) to thereby engender a suite of valid tests which can be utilized on authentic programs.

The strong and powerful principle of mutation testing is using faults that imitate mistakes that a highly confident programmer would make. To simulate this, simple syntactic changes are applied to the original program; these changes produce faulty versions of the program called mutants. If the original program and a mutant generate different outputs for a given test case, then the mutant is regarded as “dead". Therefore, the goal of mutation testing is to find a test set capable of killing a significant number of mutants.In mutation testing, mutants are able to categories into First Order Mutants (FOMs) and Higher Order Mutants (HOMs) owing to types and quantity of faults initial test case value. First Order Mutants are produced by applying a mutation operator only one time [2] whereas Higher Order Mutants [3] is produced by applying mutation operators in the programs more than one time.The remaining section of this paper is organized as follows. Section 2 briefly describes the cognate work on test case generation. Section 3 describes the proposed method and illustrate with a simple example. Section 4 concludes the paper and discusses future

Dr.Y. Kalpana, Associate Professor, VELS University, Pallavaram, Chennai, Tamil Nadu – 600117, India
Pramod Kumar Mallick, Research Scholar, VELS University, Pallavaram, Chennai, Tamil Nadu – 600117, India

work.

II. Literature Survey

Andrews et al. [16] culled eight popular C programs to compare hand-seeded faults to those engendered by automated mutation engines. The authors found the faults seeded by experienced developers were harder to catch. The authors withal found that faults conceived by automated mutant generation were more representative of authentic world faults, whereas the faults inserted by hand underestimate the efficacy of a test suite by emulating faults that would most likely never transpire. Murnane and Reed [4] illustrate that mutation analysis must be verified for efficacy against more traditional ebony box techniques which employ this technique, such as boundary value and equipollence class partitioning. The authors consummated test suites for a data-vetting and a statistical analysis program utilizing parity class and boundary value analysis testing techniques. The resulting test cases for these techniques were then compared to the resulting test cases from mutation analysis to identify redundant tests and to assess the value of any adscititious tests that may have been engendered.

The program-predicated test generation method [4, 5, 6, 7, 8, 9] research depends on either the analysis of the program's source code under test without authentically executing the program or observations on the dynamic demeanor of the software during its execution on test cases. The people aforetime mentioned are called static test generation methods, such as those taking on symbolic execution [4-9]; the latter are called dynamic methods are designated in [10] and [11]. The methods represented in [4-10] and their variants are path-oriented because the test case generation algorithms cull definitely culled paths in the program as input. In contrast, goal-oriented methods plan to achieve at executing definitely culled verbal expressions or branches in the program. The algorithm can determine the paths that cause the verbal expressions or branches to be executed.

III. Proposed System Architecture

In this section, we present a small concerning teaching example to demonstrate the basic plans and the data mutation technique related process and also describe the system architecture.

Suppose we used to test a quadratic equation program whose input having three natural numbers a, b, and c as the input value of the quadratic equation. Its function is to classify the equation into equal (two solutions are equal), or unequal (two different solution), or imaginary (i.e get imaginary value). It's based on hypothesis sqrt(b**2-(4*a*c)).

The data mutation testing process consists of an iterative sequence activities are shown in the following figure 1. The step by step process of the proposed system architecture is as follows:

Step 1: Choose Initial Test Cases Values

The data mutation testing commences with culling some initial test cases either manually or with some automatic test case generation method. These test data are called the fundamental value test cases (or seeds) because more test cases will be engendered from them. When the test cases have involute structures, it is not facile to obtain an astronomically immense and adequate set of such initial test cases. But, in our experiments have shown the method does not require a sizably voluminous number of initial test cases. A diminutive number of seeds that contain all possible types of elements of the input data will be enough.

For example, one may design three test cases for the quadratic equation program as follows to cover each type of solution with one test case.

- Test case t1: Input: (a=1, b=2, c=1), Expected output: two equal solution.
- Test case t2: Input: (a=1, b=3, c=2), Expected output: two different solutions.
- Test case t3: Input: (a=1, b=2, c=3), Expected output: Imaginary solution.

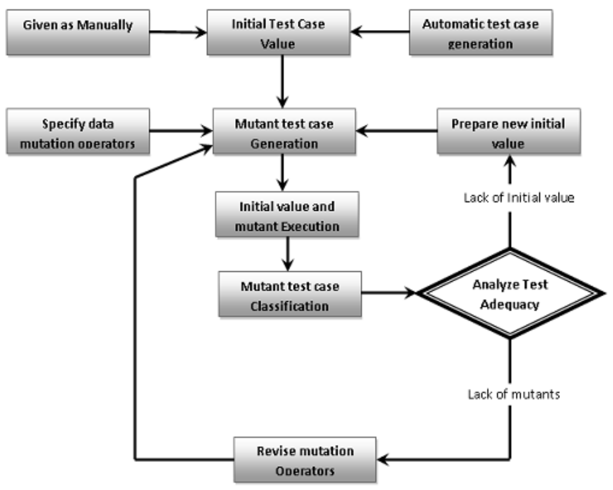
Given as Manually
Initial Test Case Value
Automatic test case generation
Specify data mutation operators
Mutant test case Generation
Prepare new initial value
Lack of Initial value
Initial value and mutant Execution
Mutant test case Classification
Analyze Test Adequacy
Lack of mutants
Revise mutation Operators

Figure 1: Proposed System Architecture

Step 2: Specifying Data Mutation Operators

Data mutation operators are simple transformations on the input value. They are required to preserve the syntactical correctness of the input data with veneration to the structure and rules on the input, if the testing is only concerned with felicitous input. Otherwise, invalid input can be engendered as well by breaking the structure or rules. Mutations' manual application can be carried out. Alternatively, the data mutation operators can be implemented in a software implement so that the generation of mutants of the initial test case value can be automated.

For testing the quadratic equation program, the organization of the input data consists of three parameters. Therefore, in order to engender test cases as sundry kinds of instances of the structure, data mutation operators should be designed to mutate the initial test case value on the parameters, such as to transmute one parameter's value by a modicum, to transmute the relationships between the parameters by interchanging (swapping) their values, etc. A rule on the valid input data is that the parameters must be natural numbers. Data mutation operators can withal be designed to test the program on both valid and invalid input data by introducing test cases that infringe the constraint.

For example, the following data mutation operators can be defined on the input data for the Quadratic Equation program.

- IVP: Increase the parameter value by 1;
- DVP: Decrease the parameter value by 1;
- SPL: Set the parameter value to a very large number, say 10000;
- SPZ: Set the parameter value to 0;
- SPN: Set the parameter value to a negative number, say -1;
- DAB: Interchange the parameter values a and b;
- DAC: Interchange the parameter values a and c;
- DBC: Interchange the parameter values b and c;
- RPL: Rotate the parameter values towards left;
- RPR: Rotate the parameter values towards right.

Step 3: Mutant Test Case Generation

Given a set of initial value and the set of categorically designed mutation operators are applied to each value to engender a set of mutants. The software implement that carries out the data mutation operations should additionally automatically locate pertinent elements in the input data for each mutation operator. The number of mutants carried out from an initial value is decided by two factors: the types and numbers of elements in the initial value and the designed data mutation operators. It is worth noting that some mutation operators can be applied to a mutant to engender other mutants, which are called the second generation mutants of the pristine test case value.

Similarly, a mutant of the second generation can additionally be acclimated to engender the third generation mutants, and so on. Whether high generation mutants should be developed and used depends on the concrete requisite of the test.For instance, by applying the mutation operator IVP to test case t1 on parameter a, we can obtain the following test case t4.Input: (a=2, b=2, c=1).

Among the above 10 data mutation operators, the first 5 can be applied on each of the 3 parameters of a seed, so thoroughly (5*3 +5)*3 = 60 test cases covering all sorts of coalescences of data elements can be systematically engendered from the three initial test case values.

Step 4: Initial Value and Mutants Execution

The initial value and their mutants are executed under software testing. On each test case, the outputs and other aspects of dynamic deportment of the software are observed and recorded for further analysis. Our approach does not depend on any concrete approach that the demeanor of the software is observed and recorded.

Step 5: Mutant Test Case Classification

The mutants can be relegated into either dead or alive according to the recorded deportment and outputs of the program under test is akin to traditional mutation testing. A particular mutant is relegated as "dead", if the software under test under execution on the mutant is different from the execution on the initial test case. Otherwise, the same mutant is relegated as "alive". Depending upon the functionality of the software under test, it varies that what precisely designates by two executions of the software on two test data are different.

For example, for a correctly implemented Quadratic Equation program, the execution on the mutant test case t4 will output imaginary solution while the execution on its initial value t1 will output two equal solutions. Therefore, the t4 test case will be dead after the test execution.

In testing other software program, relegation of mutants may be less simple as the Quadratic Equation program. For example, if the functionality of the software under test is to transform a model into executable code, the analysis of an execution of the software may involve the authentic execution of the code engendered from the pristine model as well as the execution of the code engendered from the mutants if no error message reported by the code engenderer. This may require further analysis and testing on the engendered code to distinguish them.

A conception of "dead" and "alive" mutant is different to the conception in traditional mutation testing. In traditional mutation testing, program mutants are relegated into dead or alive according to whether their deportment on a given test set is different from the pristine program. The relegation of mutants into dead and live plays a consequential role in that the percentage of dead mutants designates the fault detecting ability of the test set. However, it is less paramount in data mutation testing. Neither the aliveness nor the dead of a mutant test case betokens the program is veridical on the test case. A live mutant should be further analyzed to find the reason why the mutation of the input does not affect the output of the program under test. A dead mutant additionally needs further analysis because a difference in the demeanor of the program does not indispensably implicatively insinuate that the program deports correctly on the mutant. Nevertheless, mutation scores for data mutation testing can accommodate as utilizable bespeakers to guide further analysis of the test efficacy.

IV. CONCLUSION AND FUTURE ENHANCEMENTS

Mutants are mainly designed to be used as practical replacements for real faults in software testing research and in practice by developers. If mutation score is correlated with fault detection then this is valid one. The incipient approach presented in this paper aims at acclimating mutation analysis for building trust into incipient test cases programs utilizing this technique. The future enhancement of this paper is testing a model consistency checker will be reported later, the difference is in the checker's reports on the consistency of the models.

REFERENCE

[1] Myers, G.J., Sandler, C., Badgett, T.: (2011) "The Art of Software Testing". 3rd edition. Wiley publications.

[2] Harman, M., Jia, Y.: (2009) "Analysis and survey of the development of mutation testing". IEEE Transactions on Software Engineering journal, 649-678.

[3] Jia, Y, Harman, M, (2009) "Higher order mutation testing". Journal of Information and Software Technology, 1379-1393.

[4] Howden, W. E. (1975), "Methodology for the generation of program test data". IEEE Transactions on Computers, 554-560.

[5] Ramamoorthy, C., Ho, S. and Chen, W. (1976), "On the automated generation of program test

data". IEEE Transactions on Software Engineering, 293-300.

[6] King, J. (1975), "A new approach to program testing". International Conference on Reliable Software proceedings, Los Angeles, California, USA, 21-23 April, pp. 228-233. ACM, New York, NY, USA.

[7] Clarke, L. (1976), "A system to generate test data and symbolically execute programs". IEEE Transactions on Software Engineering, 215-222.

[8] Howden, W. E. (1977), "Symbolic testing and the DISSECT symbolic evaluation system". IEEE Transactions on Software Engineering, 266-278.

[9] Howden, W. E. (1978), "An evaluation of the effectiveness of symbolic testing". Software-Practice and Experience, 381-397.

[10] Korel, B. (1990), "Automated software test data generation". IEEE Transactions on Software Engineering, 870-879.

[11] Beydeda, S. and Gruhn, V. (2003), "BINTEST - binary search-based test case generation". 27th International Computer Software and Applications Conference proceedings (COMPSAC'03), Dallas, TX, USA, 3-6 November, pp. 28-33. IEEE Computer Society, Los Alamitos, CA, USA.

[12] Chen, H. Y., Tse, T. H. and Chen, T. Y. (2001) "TACCLE: A methodology for Object-Oriented Software Testing at the Class and Cluster Levels". ACM Transactions on Software Engineering and Methodology, 56-109.

[13] Li, S., Wang, J. and Qi, Z.-C. (2004), "Property-oriented test generation from UML state charts". 19th International Conference on Automated Software Engineering Proceedings (ASE'04), Linz, Austria, 20-25 September, pp. 122-131. IEEE Computer Society, Los Alamitos, CA, USA.

[14] Chan, K. P., Chen, T. Y. and Towey, D. (2006), "Restricted Random Testing: Adaptive Random Testing by Exclusion". International Journal of Software Engineering and Knowledge Engineering, 16, 553-584.

[15] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. (1998), "An empirical study of the effects of minimization on the fault detection capabilities of test suites". International Conference on Software Maintenance Proceedings (ICSM), pages 34–43.

[16] R. A. DeMillo, R. J. Lipton and F. G. Sayward (1978), "Hints on test data selection: Help for the practical programmer", IEEE Computer number 11, pp. 34-41.

[17] P. G. Frankl, S. N. Weiss and C. Hu (1997), "All-uses vs. mutation testing: An experimental comparison of effectiveness", Journal of Systems Software number. 38, pp. 235-253.

[18] R. M. Hierons, M. Harman and S. Danicic (1999), "Using Program Slicing to Assist in the Detection of Equivalent Mutants", Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 233-262.

[19] A. J. Offutt and J. Pan (1996), "Detecting equivalent mutants and the feasible path problem", Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press, pp. 224-236.

[20] R. T. Alexander, J. M. Bieman, S. Ghosh and J. Bixia (2002), "Mutation of Java objects," 13th International Symposium on Software Reliability Engineering, Fort Collins, CO, USA, 2002, pp. 341-351.