Oriental Journal of Computer Science & Technology Vol. **4**(1), 201-204 (2011)

Design and Analysis of New Software Conformance Testing: NA Mutation Testing

NAVEEN TYAGI[1] and ASHISH CHATURVEDI[2]

[1]Department of Computer Science, [2]Department of Applied Science & Humanities,
Gyan Bharti Institute of Technology, Meerut (India).

(Received: January 11, 2011; Accepted: February 18, 2011)

ABSTRACT

Software conformance testing is the process of evaluating the accuracy of an implementation built to the requirements of a functional specification. Tedious conformance testing of software is not practical because variable input values and variable sequencing of inputs results in so many possible combinations of tests. Mutation testing is a technique for unit testing software that, although powerful, but computationally expensive. Recent engineering advances have given us techniques and algorithms for significantly reducing the cost of mutation testing. This paper demonstrates NA (Naveen-Ashish) Mutation to design a system that will approximate mutation.

Key words: New Software, Conformance testing, Mutation testing.

INTRODUCTION

Mutation testing is a two-step process for estimating the accuracy of a given software program with respect to a given functional specification. The first step is to determine the adequacy of an existing test suite for its ability to distinguish the given program from a similar but incorrect program, i.e. a mutant of the given program. A mutation of a program is a modification of the program created by the introduction of a single, small, legal syntactic change in the software. Some mutations are equivalent to the given program, i.e given the same input they will produce the same output, and some are different. Test suite adequacy is determined by mutation analysis, a method for measuring how well a test suite is able to discover that a mutated program is either equivalent to, or different from, the original program. A test suite is 100% adequate if it is able to discover every possible non-equivalent mutation. Since 100% adequacy is nearly impossible to achieve, the second step of the mutation testing process is to make an inadequate test suite more robust by adding new test cases in a controlled manner. The test suite enhancement process stops when mutation analysis shows that the test suite has reached a desired level of adequacy. The modified test suite, together with its adequacy measure, is then used to determine a level of confidence for the correctness of the original source program with respect to its functional specification. Adequacy of a test suite with respect to a given collection of mutant programs is often defined to be the ratio of the number of discovered non-equivalent mutants from that collection divided by the total number of non-equivalent mutants in the collection. Since the number of possible mutations can be very large, often proportional to the square of the number of lines of code, testing to estimate this ratio can be a costly and time consuming process. Mutation testing and mutation analysis have matured since 1977 and researchers have found a number of ways to get equivalent results with smaller and more carefully designed classes of mutant programs. Recent researchers have shown that by choosing an appropriate class of mutations, mutation testing can be equivalent to data-flow testing or domain testing in its ability to determine program correctness. Since many of the steps in mutation analysis can be automated, the

mutation testing process is now suitable for industrial use as a white-box methodology for unit testing. Early researchers in mutation testing include A.T. Acree, T. Budd, R.A. DeMillo, F.G. Sayward, A.P. Mathur, A.J. Offutt, and E.H. Spafford. Active research centers include Yale University, the Georgia Institute of Technology, Purdue University, and George Mason University, as well as industrial software houses. Automated mutation systems include Pilot[1] at Yale University and Mothra [2] at Purdue University.

Present paper elaborates a new fundamental mutation, named Naveen-Ashish Mutation, after the names of authors.

Advantages and Disadvantages of Mutation Testing:

Advantages

Research has shown that there is a strong correlation between simple syntax errors and complex syntax errors in a program, i.e. usually a complex error will contain some simple error. By choosing a set of mutant programs that do a good job of distinguishing simple syntax errors, one has some degree of confidence (not statistically measurable) that the set of mutant programs will also discover most complex syntax errors. By proper choice of mutant operations, comprehensive testing can be performed. Research has shown that with an appropriate choice of mutant programs mutation testing is as powerful as Path testing or Domain analysis[3]. Other research has tried to compare mutation testing with Data Flow testing[4, 5] with evidence (not proof) that mutation testing is at least as powerful.

Mutation analysis is more easily automated that some other forms of testing. Robust automated testing tools have been developed at multiple universities and industrial testing groups, e.g. Yale (Pilot)[1], Purdue (Mothra)[2], Georgia Institute of Technology, and George Mason Univ. Mutation testing lends itself nicely to stochastic analysis.

Disadvantages

Each mutation is equal in size and complexity to the original program, and a relatively large number of mutant programs need to be tested against the candidate test suite. Each mutant may need to be tested against many of the test cases before it can be distinguished from the original program.

Some research has shown thata “good” set of mutant programs may require a number of mutants proportional to the square of the number of lines of code in the original program 6], potentially a very large number.

The Cost of Mutation Testing

The major computational cost of mutation testing is incurred when running the mutant programs against the test cases. Budd[1] analyzed the number of mutants generated for a program and found it to be roughly proportional to the product of the number of data references times the number of data objects. Recent empirical measurements have validated this estimate over a number of programs[10]. Typically, this is a large number for even small program units. For example, 44 mutants are generated for the function Min. Since each mutant must be executed against at least one, and potentially many, test cases, mutation testing requires large amounts of computation. This is shown in Figure below in the box labeled Run test cases on each live mutant.

```
      FUNCTION Min (I,J)
1     Min = I
   Δ  Min = J
2     IF (J .LT. I) Min = J
   Δ  IF (J .GT. I) Min = J
   Δ  IF (J .LT. Min) Min = J
3     RETURN
```

Fig. 1.

here, it should be possible to test the same procedure in 10 or 15 minutes. There are also several manual costs associated with traditional mutation systems.

In Figure 2, the solid steps are performed automatically, and the dashed steps are performed manually. The process we develop through this paper eliminates the two manual steps of inputting test cases and analyzing equivalent mutants, which dramatically reduces the human cost of applying mutation. Unfortunately, we cannot eliminate the

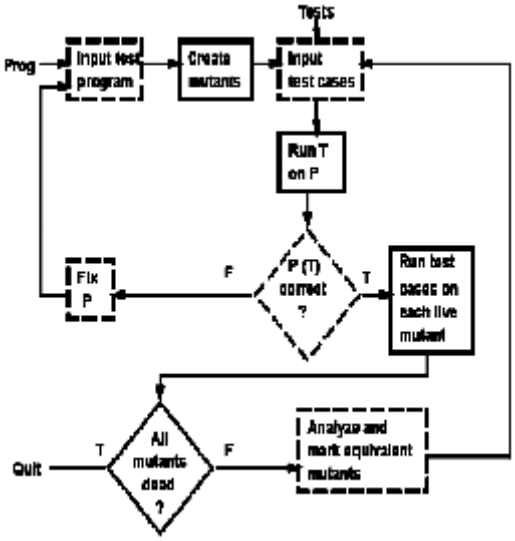

Fig. 2.

resulting major human cost, determining if the output of each test case is correct. We do however, as a result of our test data generation ability, modify the mutation process so as to reduce the number of test cases for which the programmer needs to determine output correctness.

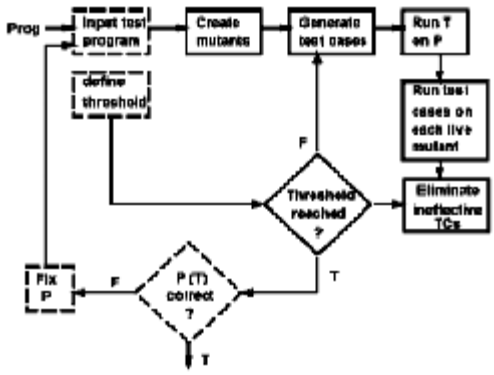

Fig. 3.

Proposed Naveen-Ashish Mutation scheme

Figure 3 presents a new model of the Naveen- Ashish (NA) mutation testing process. Initially, Godzilla will be used to generate a set of test cases (perhaps a test that is smaller than ultimately desired) and those test cases will be executed against the original program, and then the mutants. The tester will define a \threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then test cases that killed no mutants (termed ineffective), will be removed. This process will be repeated, each time generating test cases to only target live mutants, until the threshold mutation score is reached. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases, and fix the program if any faults are found.

In both the traditional and this new process, the major part of the time and effort of mutation is in the loop of generating, running, and disposing of test cases. The significant difference between the processes is that the loop in the new process contains no manual steps. All manual steps are outside the loop, and only need to be done once. In fact, the only significant manual step is that of deciding if the outputs of each test case are correct. There seems to be little hope of automating this step, although by disposing of ineffective test cases before checking outputs,we significantly reduce the workload of the tester.

REFERENCES

1. T. Budd and F. Sayward. User's guide to the Pilot mutation system, TR-114, Dept of Computer Science, Yale Univ, (1977).
2. R. A. DeMillo, D. S. Guindi, K. N. King, W.M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. Proceedings of 2nd Workshop on Software Testing, Verification, and Analysis, 142-151 (1988).
3. P. G. Frankl and E. J. Weyuker. A formal

analysis of the fault-detecting ability of testing methods. IEEE Transactions on Software Engineering, **19**(3): 202-213 (1993).

4. A. P. Mathur and W.E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. The Journal of Software Testing, *Analysis, and Verification*, **4**(1): 9-31 (1994).
5. A.J. Offut, J. Pan, K. Tewary, and T. Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. George Mason Univ ISSE technical report (1995).
6. Mehmet Sahinoglu and Eugene Spafford. Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing, Purdue Univ, SERC TR-79-P, (1990).
7. A. J. Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. Technical report ISSE-TR- 94-100. Department of Information and Software Systems Engineering,George Mason University, Fairfax VA (1994).