On reduced neighborhood mutation analysis using a single mutagenic operator

Roland H. Untch
Department of Computer Science
Middle Tennessee State University
Murfreesboro, TN 37132–0048
(615) 898-5047
untch@mtsu.edu

ABSTRACT

Mutation analysis evaluates the quality of a test suite based on its ability to reveal simple faults, termed *mutations*, injected in the program under test. Unfortunately, the need to execute many slightly different versions, or *mutants*, of a program makes mutation analysis a computationally expensive technique. This paper reports on a promising approach that reduces the cost of mutation analysis by reducing the number of mutants that need to be executed.

Keywords

software testing, mutation analysis, constrained mutation, sufficient mutant operators, reduced neighborhood mutation

1. INTRODUCTION

Formal testing techniques establish *test data adequacy criteria* that seek to measure the quality of the test data used to exercise a given program; we use *program* to denote the software under test, which may be a complete program or a smaller unit. When using a test data adequacy criterion, testing ceases when either the set of test cases meets a minimum quality goal imposed by the criterion or an incorrect output is observed.

Mutation-based software testing, or *mutation testing*, uses an adequacy criterion. In mutation testing, the test set is analyzed to determine a quality measure called the *mutation adequacy score*; this assessment process is called *mutation analysis*. Mutation analysis requires executing many slightly different versions, or mutants, of a program. The computational expense of generating and running vast numbers of mutant programs is large. Approaches to reduce this computational expense usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*. The *do fewer* approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The *do smarter* approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs. The *do faster* approaches focus on ways of generating and running each mutant program as quickly as possible. This paper investigates a promising new *do fewer* approach of performing mutation analysis.

Section 2 introduces mutation analysis. Section 3 reviews related work. Section 4 presents SOMETHING. Section 5 presents our preliminary conclusions and discusses future work.

2. MUTATION ANALYSIS

Mutation analysis is a *white-box* testing technique that measures the quality of a test set by its ability to differentiate the program under test from a set of marginally different, and presumably incorrect, alternate programs [6, 7]. A test case differentiates two programs if it causes the two programs to produce different results.

The process of performing mutation analysis on a test set, T, relative to a given program, P, begins by executing P against every test case in T. If P computes an incorrect result, the test set has fulfilled its obligation and the program must be changed. Determining the correctness of these results is called the *oracle* problem [18]; this problem is common to all testing techniques and will not be discussed further. If, however, P computes correct results for every test case in T, a set of alternate programs, or *mutants*, is produced by subjecting P to a series of modification rules, G, called *mutagenic operators* or *mutagens*[1]. Each mutant program, P_i, differs from P in a simple, syntactically correct way. The syntactic change itself is called the *mutation*. The original program plus the mutant programs are collectively known as the *program neighborhood*, N, of P [3]. Figure 1 illustrates these concepts.

The next step in performing mutation analysis is to execute each mutant against the test cases in T. If, for some test case in T, a mutant produces a result that differs from the original program, we say that test case has *killed* the mutant, indicating that the test case detected the fault contained within the mutant. Once killed, these *dead* mutants are not executed against any additional test cases.

Equivalent mutants are syntactically different from, but functionally identical to, the original program. Because no test case can kill these equivalent mutants, they must not

[1]The terminology varies [20]; they are also sometimes called *mutant operators*, *mutation operators*, *mutation transformations*, and *mutation rules*.

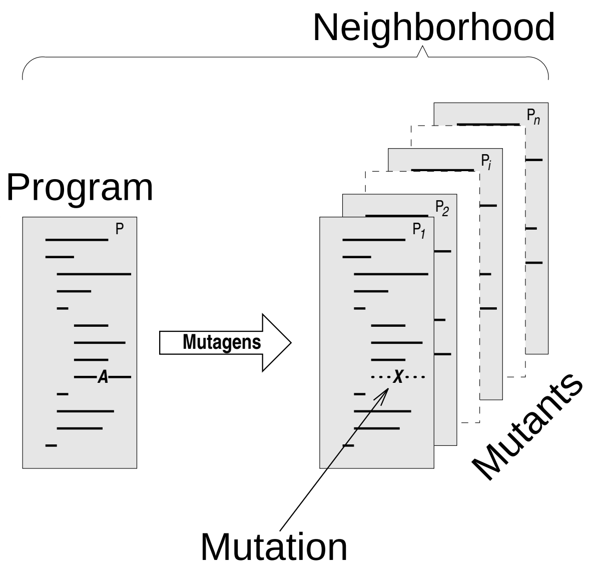

Figure 1: Components of a program neighborhood.

be considered in assessing test data quality. Although determining equivalence is in general undecidable [9], heuristic techniques for automatically identifying equivalent mutants continue to improve [8].

The ratio of dead mutants to the remaining undifferentiated *live* mutants indicates the quality of the test set. This quality is expressed by the *mutation adequacy score*, *MS*, which is the percentage of potentially killable mutants that have been killed by T, or

$$MS_G(P,T) = \frac{\#Dead}{\#Mutants - \#Equivalent} \times 100\%$$

where $\#Mutants$ is the total number of mutants in the program neighborhood. We subscript the mutation adequacy score *MS* by the set of mutagenic operators G to reflect their influence on the number and type of mutants produced. In the literature it is common to omit this subscript when a standard set (for the C language, see [1]) of mutagenic operators is understood to be in use.

The major computational cost of mutation analysis is incurred when executing the mutant programs against the test cases. With a standard set of mutagenic operators, the number of mutants generated for a program is proportional to the product of the number of data references and the number of data objects [15], which is typically a large number. For example, 4937 mutants are generated for the 138 line `tcas` program, the smallest benchmark program used in this study.

3. RELATED WORK

There have been a number of attempts to lower execution costs by running only a subset of the mutants—the *do fewer* strategy. We use *reduced neighborhood* mutation to refer to any approach that uses a reduced set of mutants; in contrast, *standard neighborhood* mutation refers to approaches that use a standard set of mutants. The concern, of course, is that the mutation adequacy score derived from analyzing a reduced neighborhood should be a good estimate, or prediction, of what the mutation adequacy score would be if the standard neighborhood had been used.

Drawing a random sample of the mutants in a program neighborhood has been proposed in several forms [2, 17]. Each sampling scheme, however, has had some drawbacks [10]. Save to say that sampling is orthogonal to the approach discussed in this paper, it will not be discussed further.

Another way of obtaining a reduced program neighborhood is to restrict the set of mutagenic operators used in generating the neighborhood. However it is not obvious which reduced set of operators should be used. In *constrained mutation*, as proposed by Mathur [12], those mutagenic operators thought to be most significant in fault detection are selected. In Wong's investigation of constrained mutation [19], only two operators (`VDTR` and `ORRN`)[2] are applied to the program to generate a reduced neighborhood. In *selective mutation*, as proposed by Offutt et al. [16], mutagenic operators that produce a large number of mutants are excluded. For *expression-selective (E-selective) mutation* [15], only six expression modification mutagenic operators are used[3]: `VDTR`, `OAAN`, `OBBN`, `ORRN`, `OLNG`, and `VTWD`.

Interface Mutation (IM) is an extension of mutation testing proposed by Delamaro et al. [5] that introduces additional mutagenic operators that model integration errors. These IM operators can be used separately or in conjunction with conventional unit operators to create a program neighborhood. The *Proteum/IM* mutation analysis system [4], used to perform our mutation analyses, incorporates both interface and unit operators when creating mutants. **We shall consider a standard program neighborhood to be that created with both IM and unit operators.** There are a total of 108 operators in the set of mutagenic operators, G, implemented by *Proteum/IM.*

4. ESTIMATING STANDARD NEIGHBORHOOD MUTATION ADEQUACY SCORES USING A REDUCED NEIGHBORHOOD ANALYSIS

Namin et al. [13, 14] have used statistical procedures to identify several *sufficient sets* [15] of C mutagenic operators. A sufficient set, R, is a proper subset $R \subset G$ such that $|R| \ll |G|$ yet $MS_G(P,T) \simeq MS_R(P,T)$. In other words, mutation analysis done on a reduced neighborhood generated from R should give roughly the same results as analysis done on a standard neighborhood. Because the reduced neighborhood is smaller, the computational cost of performing the analysis is smaller; there will be fewer mutants to execute.

An important issue specifically addressed by Namin, that was not considered in previous studies, is the question of how well the estimated (or predicted) mutation adequacy score drawn from the reduced program neighborhood corresponds to the standard score *over all ranges of the score.* Previous

[2]Actually Wong's investigation used the FORTRAN mutagenic operators `ABS` and `ROR`; the C language equivalents are cited here for consistency.

[3]Offutt et. al.'s study selected five FORTRAN mutagenic operators: `ABS`, `AOR`, `LCR`, `ROR`, and `UOI`. The C language equivalents are once again cited for consistency; the FORTRAN `UOI` operator embodies two C language operators (`OLNG` and `VTWD`), hence the difference in the number of operators.

Table 1: Siemens suite of benchmark programs

Program Name	LOC	Functions	Description
print_tokens1	402	21	lexical analyzer
print_tokens2	483	20	lexical analyzer
replace	516	21	pattern replacement
schedule	299	18	priority scheduler
schedule2	297	16	priority scheduler
tcas	138	8	altitude separation
tot_info	346	16	information measure

studies concentrated on mutation scores near 100%. The work of Namin looks at the entire range of possible scores; indeed, many of the statistical procedures employed would not otherwise work.

Although Namin identified several sufficient sets of operators, one particular set was selected as the best. This set of 28 mutagenic operators includes five interface (IM) operators { `IndVarAriNeg`, `IndVarBitNeg`, `RetStaDel`, `ArgDel`, `ArgLogNeg` } and 23 unit operators { `OAAN`, `OABN`, `OAEA`, `OALN`, `OBBN`, `OBNG`, `OBSN`, `OCNG`, `OCOR`, `Oido`, `OLAN`, `OLBN`, `OLLN`, `OLNG`, `OLSN`, `ORSN`, `SGLR`, `SMTC`, `SMVB`, `SSWM`, `STRI`, `SWDD`, `VGPR` }. All reduced neighborhoods identified as "Namin" were generated using this set of 28 operators.

Namin used a standard set of programs, commonly encountered in the software testing literature, to mutation analyze. The Siemens suite of benchmark programs (originally assembled by Hutchins et al. [11]) are much bigger than those used in many of the previous reduced neighborhood studies. See Table 1. Although many programmers would consider these programs small, they are well accepted as specimens in the software testing community. Even as small as they are, Namin observed that hundreds of hours of computer time were needed to analyze them.

Namin measured test suite effectiveness relative to non-equivalent mutants; that is, using the standard definition of the mutation adequacy score given above. However most software testers use a *raw* mutation adequacy score when initially assessing the quality of their test data. This so-called raw score does not subtract out the number of equivalent mutants. This is done because determining *a-priori* which are the equivalent mutants is incredibly difficult.

Our work began as a replication study of Namin's work with two changes: (1) raw mutation scores would be used in the statistics because that is what is usually observed in the state-of-the-practice, and (2) stratified sampling of test suites would be introduced. In Namin's studies, the data collection was weighted heavily with test suites that produced high mutation adequacy scores. Because the Siemens test cases were designed from the outset to be "good" at testing a program, a random selection of these test cases would naturally result in skewed test suite scores. Stratified sampling reduces sampling error by providing a more uniform distribution of possible scores.

For each of the Siemens programs a total of 201 test suites were analyzed and mutation adequacy scores calculated for five different types of neighborhoods. We generated 100 test suites by randomly selecting from the universe of test cases two of each size from one test case to 50 test cases. Another 100 test suites consisted of a single test case randomly drawn by strata. The final test suite for each program was comprised of the entire pool of test cases and represented a

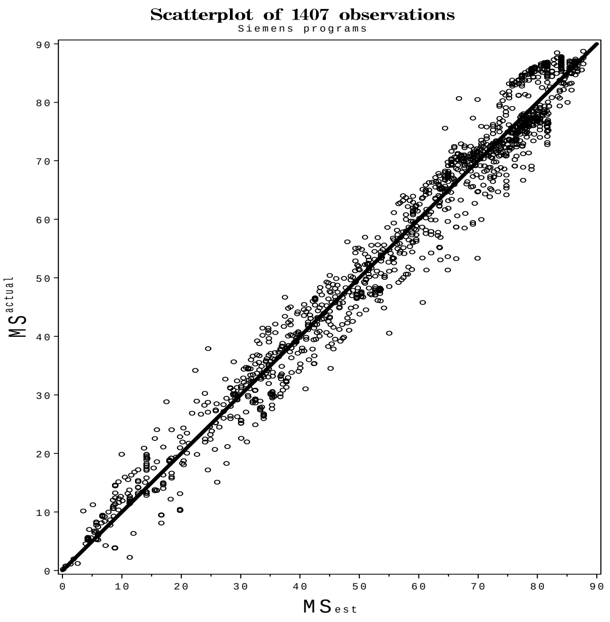

Figure 2: Actual vs. `SSDL` estimated mutation scores.

maximum case.

The five different neighborhoods were formed as follows. One neighborhood was the standard program neighborhood resulting from applying the full set of mutagenic operators supported by *Proteum/IM*. Three reduced neighborhoods were produced by using the operator sets of Wong, Offutt, and Namin, respectively. The final reduced neighborhood was formed by using the single mutagenic operator `SSDL`. The `SSDL` operator systematically deletes each statement, one at a time, from P to produce mutants; the number of mutants produced is linearly proportional to the number of lines of code. The sizes of the respective neighborhoods are shown in Table 2.

Using the SAS statistical package, a regression analysis was done for each of the four reduced neighborhood models to determine how well the mutation score estimate predicted the standard mutation score. For each model, the coefficient of determination was calculated. The *coefficient of determination*, or R^2 value, for a model is defined as the proportion of the variability in the predicted, or dependent, variable (in this case, standard *MS*) that can be accounted for by the explanatory variable (in this case, reduced *MS*) of the model. The higher the R^2 value, the better the prediction. The R^2 values for the four models was found to be: Wong = 0.84, Offutt = 0.92, Namin = 0.96, and `SSDL` = 0.97. A plot of actual (standard) *MS* versus the corresponding `SSDL` *MS* scores is given in Figure 2.

5. CONCLUSIONS AND FUTURE WORK

We are not surprised that the preliminary results of our replication study should bear out that the Namin operator set is a good choice for reduced program neighborhood mutation analysis. However that using the single `SSDL` operator produces comparable results was not expected and suggests considerably more investigation. As can be see from Table 2. the `SSDL` operator generated only between 1.3% to 4.3% the mutants compared to standard neighbor-

Table 2: Comparison of reduced neighborhood strategies

Program Name	*#Mutants* Unit	*#Mutants* IM	*#Mutants* Std. Total	*#Mutants* *Wong*	*#Mutants* *Offutt*	*#Mutants* *Namin*	*#Mutants* `SSDL` (%Std.)
print_tokens1	4262	7180	11442	386	606	984	263 (2.3%)
print_tokens2	4607	5433	10040	452	634	906	298 (3.0%)
replace	11576	13354	24930	1143	1899	1629	336 (1.3%)
schedule	2398	1877	4275	238	385	339	183 (4.3%)
schedule2	3062	3610	6672	300	478	587	183 (2.7%)
tcas	2872	2065	4937	186	315	371	70 (1.4%)
tot_info	6456	2460	8916	650	1115	647	126 (1.4%)

hood mutation. Such small percentages would translate into big performance gains. Moreover, a mutation analysis tool based solely on the `SSDL` operator would be straightforward to build.

We are in the process of mutation analyzing a much larger program specimen, `space`, with a standard neighborhood of 316449 mutants. Once the data from this program is available, we intend to see if the promise of this discovery bears out with this bigger program.

6. REFERENCES

[1] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, Mar. 20 1989.

[2] T. A. Budd. *Mutation Analysis of Program Test Data.* Ph.D. dissertation, Yale University, New Haven, CT, May 1980.

[3] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Inf.*, 18(1):31–45, Nov. 1982.

[4] M. E. Delamaro and J. C. Maldonado. *Proteum/IM 2.0*: An integrated mutation testing environment. In *Mutation testing for the new century*, pages 91–101, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[5] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Softw. Eng.*, 27(3):228–247, 2001.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[7] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, SE-3(4):279–290, July 1977.

[8] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability*, 9(4):233–262, Dec. 1999.

[9] W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, NY, 1987.

[10] W. Hsu, M. Şahinoğlu, and E. H. Spafford. An experimental approach to statistical mutation-based testing. Technical Report SERC-TR-63-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, Apr. 10 1990.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[12] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC)*, pages 604–605, Tokyo, Japan, Sept. 11–13 1991. IEEE Computer Society Press.

[13] A. S. Namin and J. H. Andrews. Finding sufficient mutation operators via variable reduction. In *MUTATION '06: Proceedings of the Second Workshop on Mutation Analysis*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

[14] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 351–360, New York, NY, USA, 2008. ACM.

[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, Apr. 1996.

[16] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 17–21 1993. IEEE Computer Society Press.

[17] M. Şahinoğlu and E. H. Spafford. A Bayes sequential statistical procedure for approving software products. In W. Ehrenberger, editor, *Proceedings of the IFIP Conference on Approving Software Products (ASP–90)*, pages 43–56, Garmisch-Partenkirchen, Germany, Sept. 1990. Elsevier/North Holland, New York.

[18] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, Nov. 1982.

[19] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, Dec. 1995.

[20] D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell. A practical method for software quality control via program mutation. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 159–170, Banff, Alberta, Canada, July 19–21 1988. IEEE Computer Society Press.