Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing [*]

SERC TR–79–P

Mehmet Şahinoğlu [†]
Department of Statistics
Mathematical Sciences Building
Purdue University
West Lafayette, Indiana 47907
xnse@vm.cc.purdue.edu

Eugene H. Spafford
Software Engineering Research Center
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
spaf@cs.purdue.edu

10 May 1990

Abstract

Mutation analysis is a well-studied method of measuring test-case adequacy. Mutation analysis involves the *mutation* of a program by introduction of a small syntactic change in the software. Existing test data sets are then executed against all these *mutant programs*. If the test data set is adequate for testing the original program, it will distinguish all of the incorrect mutant programs from the original program. As an ad-hoc procedure, a stopping criterion is conventionally based on a given "Y% of the mutants to be distinguished" with a certain "confidence level of X%" over a multiplicity of random test cases.

Alternatively, we propose a Bayes sequential procedure for testing $H_0 : p = p_1$ (acceptable fraction of live mutants to demonstrate good quality) vs. $H_A : p = p_2$ (unacceptable fraction of live mutants to demonstrate bad quality). This derives a sequential probability ratio testing (SPRT) that is the most economical sampling scheme with given prior probabilities, decision and sampling cost functions. The implementation of our proposed method on a sample program shows the cost effectiveness of the new technique as compared to the current, deterministic approach, which was not structured by statistical hypothesis testing.

[*] To appear in Proceedings of the IFIP Conference on Approving Software Products (ASP-90), Garmisch-Partenkirchen, Germany, 17 September 1990.

[†] Currently visiting Purdue University from Middle East Technical University (METU), Ankara, Turkey on a Fulbright Scholarship, support which is gratefully acknowledged.

1 Introduction

Software testing is a critical phase of the software development life cycle. However, software testing can be an expensive proposition. One desires to obtain an adequate measure of the reliability of a software system while minimizing the expenditure of resources. Testing is always a trade-off between increased confidence in the correctness of the software under examination, and constraints on the amount of time and effort that can be spent testing that software.

For over a decade, researchers have been working with *program mutation* as a method of developing test cases to test software.[1] A basic goal of program mutation is to provide the user with a measure of test set *adequacy*, by executing that test set against a collection of program *mutations*. Mutations are simple changes introduced one at a time into the code being tested. These changes are derived empirically from studies of errors commonly made by programmers when translating requirements into code, although theoretical justification also can be found for their selection[12].

A mutant is *killed* if the execution of the mutated code against the test set distinguishes the behavior or output of the mutation from the unmutated code. The more mutants killed by a test set, the better the measured adequacy of the test set. By proper choice of mutant operators, comprehensive testing can be performed, [5] including path coverage [14] and domain analysis. [23] By examination of unkilled mutants, testers can add new test cases to better the adequacy score of the entire test set.

Mutation analysis is designed to substantiate the correctness of a program Φ. Mutation analysis strives to develop test data that, when applied to the program Φ, illustrates the equality of the *intended* and *observed* behaviors of Φ. This methodology is normally embedded in a test environment that enables a tester to test his program interactively.

Somewhat more formally, the mutation approach is to induce syntactically correct changes into a program Φ, thereby creating a set of *mutant programs*. Each mutant represents a possible error in Φ, and the goal of the tester is to construct a set of test data τ that distinguishes the output or behavior of $\Phi(\tau)$ from that of all mutant programs. Test data sensitive enough to distinguish all mutant programs is deemed adequate to infer the probable correctness of Φ.

The mutation analysis methodology is as follows: submit a program Φ and a set of test data τ whose adequacy is to be determined. The mutation system first executes Φ with respect to τ. If the results are incorrect, then certainly Φ is in error. However, if the results appear correct, it may still be that Φ is in error, for the test data set τ may not be adequate to distinguish this inherent incorrectness. In this case, a set of *mutant programs* are created, call them $\Phi_1, \ldots, \Phi_m$. Each mutant program differs from the original program Φ by one single-point, syntactically correct change. Such *mutant transformations* are statically defined for a language Λ and designed to expose errors frequently committed by programmers using Λ.

For each execution of a mutant against the test set, $\Phi_i(\tau)$, exactly one of two things happens:

[1] The mutant $\Phi_i(\tau)$ yields different results than $\Phi(\tau)$, or

[1] Program mutation has been well documented in the literature and will only be summarized here. The reader unfamiliar with mutation testing is directed to recent references on mutation for detailed descriptions and further references, e.g. [8, 21, 13, 6, 24, 16].

[2] The mutant $\Phi_i(\tau)$ yield the same results as $\Phi(\tau)$.

In the first case, the mutant is said to be *dead* since the mutant difference between Φ_i and Φ has been distinguished by τ. In the second case, either:

[a] τ is not adequate to distinguish the mutant that gave rise to Φ_i, or

[b] Φ and Φ_i are *equivalent programs* and no test data τ can distinguish their behavior (the "error" that gave rise to Φ_i was not an error at all, but an alternate encoding of Φ). Such a Φ_i is called an *equivalent mutant.*

Test data that leaves no live mutants or only equivalent mutants is said to be of adequate sensitivity to infer the probable correctness of Φ. Moreover, the ratio of dead mutants to the total number of nonequivalent mutants yields a relative measure of the adequacy of the test data τ in testing Φ. The test set τ may be augmented with additional test cases and the process repeated until an appropriate level of test case adequacy or threshold level in the cost of the test (in terms of dollars, resources, etc.) has been reached. The test set τ can then be carried into the maintenance stage of the software life cycle.

2 On MOTHRA and the Stopping Rule

The MOTHRA software testing environment [10, 9, 8] is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis, and evaluation of mutation-based tests of software systems. MOTHRA is designed to be used starting at the earliest stages of software development and continuing through the progressively later stages of system integration, acceptance testing, operation, and maintenance. It has been in use at Purdue and other locations for the last few years as a testbed for experimental work in software engineering.

One of the drawbacks to full mutation testing with a system such as MOTHRA involves the number of mutations that must be executed. Each mutation is equal to the size and complexity of the original program, and the number of mutants is proportional to the square of the number of lines of source code present in the original software. Each mutant may need to be run against many test cases before being distinguished from the original code. For routines in the hundreds of lines of code, this may result in millions of separate executions.

It would be beneficial to test only a small statistically random sampling of mutants against the given test cases to determine, at some arbitrary level of confidence, that the program is correct. This would be of great utility in situations where full mutation testing would be too expensive, too time consuming, or simply beyond the capabilities of the available testing software.

Conventionally, a typical mutation system user must specify test requirements of the following form: [13] "I must be X% sure that Y% of the mutants of type Z are killed in unit A." Here, X% is the degree of assurance MOTHRA will use to perform the random selection of mutants of type Z in unit A. The user must still monitor the status information to determine whether or not the stopping criteria Y% for the test of mutants of type Z in unit A has been achieved. Hence, one defines a stopping criterion based on a given "Y% of the mutants to be killed" with a certain confidence level X% over a multiplicity of test-cases.

This approach may be defined as a deterministic stopping rule-of-thumb, since the entire experiment is of a random nature and is not structured by statistical hypothesis testing. The customary ad hoc approach, although straightforward and easy, may result in a waste of testing time and resources due to lack of a statistical risk analysis. The only definitive statement that can be made from such testing involves achieving a 100% level of mutant distinction.

3 Sequential Statistical Procedures

The usual statistical hypothesis-testing procedures ordinarily have fixed sample size. In particular, the statistical testing procedures have only answered the question, "Do we have sufficient evidence to declare a particular hypothesis false?" Other models address the question, "When have we corrected an 'optimum' number of errors? That is, should we now release the software?"[2] However, if we allow the sample size to increase without limit, we can obtain a test for any prespecified probability of occurrence of Type I and Type II errors.[2] Thus, if testing for $H_0 : \Theta = \Theta_1$ vs $H_1 : \Theta = \Theta_2$, we can assure ourselves of our choice given the probability of a wrong decision. We can thus test software to any prespecified level of confidence and criticality, rather than to some single, theoretical "optimum" point.

Our approach will be to sample $X_1 = x_1, \ldots, X_k = x_k$ and after each observation $X_k = x_k$, make a decision based on $x_1, \ldots, x_k$ whether or not to continue sampling. If we stop, we want to choose between H_0 and H_1.

Let $X_1, \ldots, X_n$ be a sequence of i.i.d. (identical independently distributed) random variables (r.v.) with p.d.f. (probability distribution function) $F(x; \Theta), \Theta \in \Omega$(parameter space), where the x_i's are observed sequentially. Let $\mathcal{A}$ be the space of actions available to the statistician.

Definition: A sequential decision procedure has two components:

1. A stopping rule which specifies for every set of values $(x_1, x_2, \ldots, x_n), n \geq 1$ whether to stop sampling and choose a decision in $\mathcal{A}$, or to continue sampling and take another observation x.
2. A decision rule $d_n(x_1, \ldots, x_n)$ which chooses the action in $\mathcal{A}$ to be taken for the set of values $(x_1, \ldots, x_n)$ when the sampling is stopped.

4 The Sequential Probability Ratio Test (SPRT)

Let $x_1, x_2, \ldots$ be i.i.d r.v.'s with p.d.f. f(x;Θ) $\Theta \in \Omega = \{\Theta_1, \Theta_2\}$. Let $f_1(x) = f(x; \Theta_1); f_2(x) = f(x; \Theta_2)$. We want to test $H_0 : \Theta = \Theta_1$ vs $H_1 : \Theta = \Theta_2$ without fixing the sample size in advance.

For a fixed sample size k, the NP (Neyman Pearson) Lemma tells us to reject for large values of the ratio

[2] In this context, a Type II error is mistakenly accepting a faulty program. A Type I error is mistakenly rejecting a good program as faulty.

$$\lambda_k = \frac{f_2(x_1)f_2(x_2)\dots f_2(x_k)}{f_1(x_1)f_1(x_2)\dots f_1(x_k)} \tag{1}$$

Wald [20] tells us to consider the sequence of ratios $\lambda_1(x_1), \lambda_2(x_1, x_2), \dots, \lambda(x_1, \dots, x_k)$, i.e. consider the sequence:

$$\frac{f_2(x_1)}{f_1(x_1)}, \frac{f_2(x_1)f_2(x_2)}{f_1(x_1)f_1(x_2)}, \frac{f_2(x_1)\dots f_2(x_k)}{f_1(x_1)\dots f_1(x_k)} \tag{2}$$

Definition: The SPRT [22, 17, 11] (Test) for testing $H_0 : \Theta = \Theta_1$ versus $H_1 : \Theta = \Theta_2$ is a rule that states:

1. if $\lambda_k(x) >= \mathcal{A}$, stop sampling and reject H_0 (accept H_1).
2. if $\lambda_k(x) <= \mathcal{B}$, stop sampling and reject H_1 (accept H_0).
3. if $\mathcal{B} < \lambda_k(x) < \mathcal{A}$, continue sampling and take another observation x_{k+1}.

Here $\mathcal{A}$ and $\mathcal{B}$ are constants determined so that α = P{Type I error} = P{Reject $H_0 \,|\, \Theta_1$}, i.e. probability of rejecting H_0 when $H_0 : \Theta = \Theta_1$ is true. β = P{Type II error} = P{Fail to Reject $H_0 \,|\, \Theta_2$}, i.e. probability of accepting H_0 when H_1 is true.

Namely, if N is the stopping time for this procedure,

$$\alpha = P_{\Theta_1}(\lambda_N(x) >= \mathcal{A}) \tag{3}$$
$$\beta = P_{\Theta_2}(\lambda_N(x) <= \mathcal{B}) \tag{4}$$

5 Binomial SPRT

Let $x_1, x_2, \dots$ be i.i.d. Bernoulli r.v.'s with p.d.f. f(x;Θ) = $\Theta^x(1-\Theta)^{1-x}$ x=0,1.

We now wish to test for $H_0 : \Theta = \Theta_1$ vs $H_1 : \Theta = \Theta_2$.

$$z_i = \log(\frac{f(x_i;\Theta_2)}{f(x_i;\Theta_1)}) = \begin{cases} \log(\frac{\Theta_2}{\Theta_1}) & \text{if } x_i = 1 \\ \log(\frac{1-\Theta_2}{1-\Theta_1}) & \text{if } x_i = 0 \end{cases}$$

Then

$$S_k = \sum_{i=1}^{k} Z_i = r_k \log(\frac{\Theta_2}{\Theta_1}) + (k - r_k)\log(\frac{1-\Theta_2}{1-\Theta_1}) \tag{5}$$

where r_k = number of 1's in $x_1, \dots, x_k = \sum_{i=1}^{k} x_i$. We observe $S_k = s_k$ and continue sampling if $b < s_k < a$ i.e.

$$b < r_k \log(\frac{\Theta_2}{\Theta_1}) + (k - r_k)\log(\frac{1-\Theta_2}{1-\Theta_1}) < a, \tag{6}$$

$$b < r_k(\log(\frac{\Theta_2}{\Theta_1}) - \log(\frac{1-\Theta_2}{1-\Theta_1})) + k\log(\frac{1-\Theta_2}{1-\Theta_1}) < a \tag{7}$$

Solving the inequality for r_k, we obtain, as illustrated in figure 1:

$$\frac{b-k\log(\frac{1-\Theta_2}{1-\Theta_1})}{\log(\frac{\Theta_2}{\Theta_1})-\log(\frac{1-\Theta_2}{1-\Theta_1})} < r_k < \frac{a-k\log(\frac{1-\Theta_2}{1-\Theta_1})}{\log(\frac{\Theta_2}{\Theta_1})-\log(\frac{1-\Theta_2}{1-\Theta_1})} \quad \text{if } \Theta_2 < \Theta_1 \tag{8}$$

$\alpha = \beta = 0.100,\ \Theta_1 = \frac{1}{4},\ \Theta_2 = \frac{3}{4}$

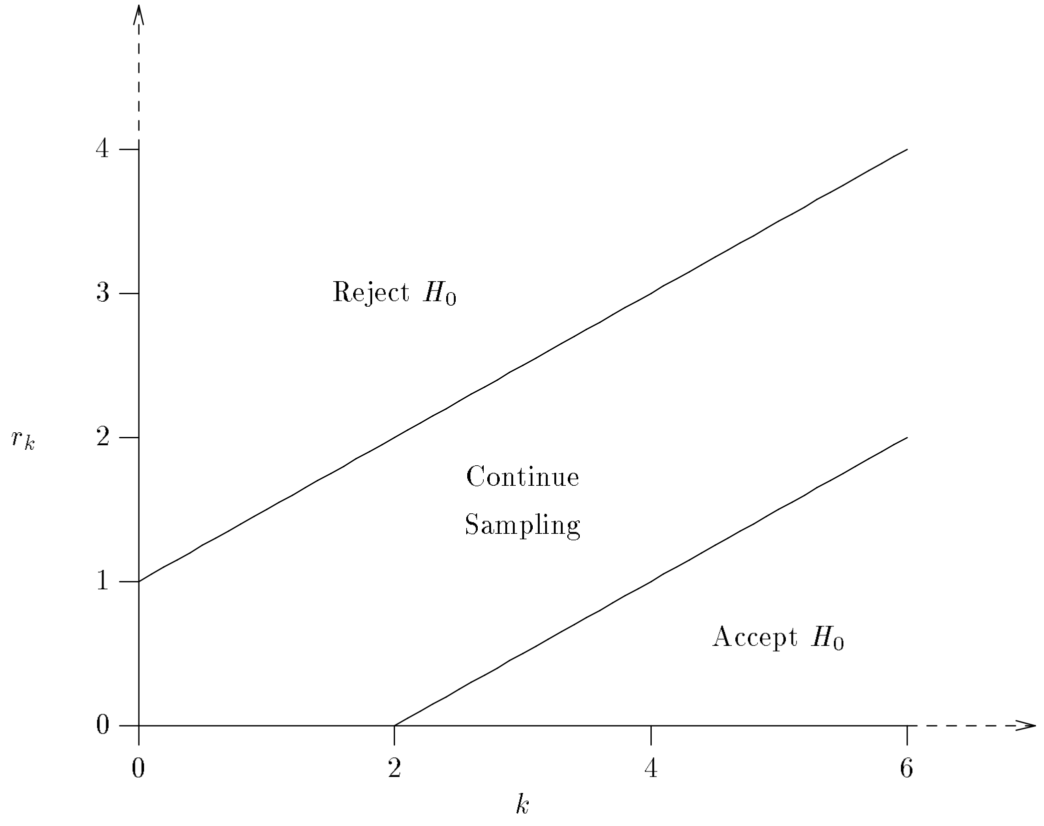

Figure 1: $r_k = \sum_{i=1}^{k} x_i$ vs. k

6 An Economical Bayesian SPRT in Mutation-Based Testing

We now propose a Bayes sequential procedure for testing $H_0 : p = p_1$ (acceptable fraction of live mutants to demonstrate good quality) vs. $H_A : p = p_2$ (unacceptable fraction of live mutants to demonstrate bad quality), where the prior distribution is given as $P(p = p_1) = a_1, P(p = p_2) = a_2; a_1 + a_2 = 1, p_1 < p_2$.

This derives a sequential probability ratio test (SPRT) that is the most economical sampling scheme with given prior probabilities, and decision and sampling cost functions. The implementation of our proposed method on a sample program shows the cost effectiveness of the new technique as compared to the current, deterministic approach, which was not structured by statistical hypothesis testing. We note here that Bayesian Sequential testing also defaults to the simple sequential procedure with no priors; or, if used with priors, when the sample size increases the a effect eventually becomes negligible. [11]

In a mutation-based testing environment, a sample (batch) of mutants (items) is inspected (executed against a test data set) and the mutants are classified as either dead (effective) or live (defective). Let p denote the fraction of live mutants within a selected sample of nonequivalent mutants. We assume that:

- a critical fraction of live mutants, p_0, is known to exist;
- that a sample with a fraction of live mutants $p < p_0$ is accepted as a good sample (the program is judged correct); and
- others with $p > p_0$ are programs not yet judged correct, and are returned for further testing or rejected.

Let $P(p)$ denote the prior distribution function based on apriori subjective software engineering judgment. Within the sample for the software product submitted for inspection, we have the following: $P(p = p_1) = a_1, P(p = p_2) = a_2$ where $a_1 + a_2 = 1$ and $p_1 < p_0 < p_2$.

Also, let W_{21} denote the decision cost incurred if a software product with fraction live-mutants p_1 is rejected, analogous to the type-I probability of α in rejecting a good product mistakenly. Similarly, let W_{12} denote the decision cost incurred if a software product with a fraction live-mutants p_2 is accepted, analogous to the type-II probability of β in accepting a bad product mistakenly. Let c denote the cost of inspecting a single mutant. Now, the problem is to determine the most economical sampling inspection plan. We want to structure our problem of product acceptance sampling to that of testing two simple hypotheses $H_0 : p = p_1$ vs. $H_A : p = p_2$. Namely, we decide to accept a software product if H_0 holds true, and to reject it if H_A holds true. Otherwise, we continue sampling.

In a sequential sampling inspection plan, one mutant (item) is inspected at a time and the inspection is stopped as soon as sufficient evidence is observed in favor of either of the hypotheses. Hence, as long as the cost of inspection depends merely on the total number of mutants inspected and no extra overhead cost is involved, a sequential sampling inspection plan will be the most economical one. Wald and Wolfowitz [19] proved that when testing two simple hypotheses, the sequential probability ratio test requires, on average, the fewest observations among all tests with the same power.

Thus, this means that of all the sampling plans with the same or lower decision costs, a plan based on the SPRT will have the minimum cost of inspection. For mutation testing, where each test involves potentially expensive execution of mutated programs against large test sets, this finding is especially important.

Following is the method of arriving at the optimum SPRT procedure based on the basic theory given by Barnard [3]:

The optimum test procedure will be given as:

- Continue inspection as long as $\lambda_2 < \lambda = (a_1/a_2)l(x, y) < \lambda_1$
- Stop inspection and accept the batch as soon as $\lambda > \lambda_1$

- Stop inspection and reject the batch as soon as $\lambda < \lambda_2$

where x and y denote, respectively, the number of dead mutants and live mutants obtained at any stage, and $l(x, y)$ is the likelihood ratio. This ration is equal to: $l(x, y) = (p_1/p_2)^y(q_1/q_2)^x$ where $q_1 = 1 - p_1$ and $q_2 = 1 - p_2$.

It is more convenient to write the inequality in step 1 of the above as follows:

$$L = \frac{a_2\lambda_2}{a_1} < l(x, y) < U = \frac{a_2\lambda_1}{a_1} \tag{9}$$

in which similar rules for acceptance and rejection apply.

Let $A(U, L; p)$ be the probability that a sample with fraction live-mutants p will be accepted. Let $R(U, L; p) = 1 - A(U, L; p)$ be defined as the rejection probability. Let $S(U, L; p)$ denote the average number of mutants required to be inspected.

To obtain equations for the optimum boundaries, we proceed in the manner indicated in equation (9). [18] The decision boundary is the locus of points such that the expected cost of taking an immediate decision is equal to the expected cost of taking at least one more observation and continuing the test. For instance, when $\lambda = \lambda_1$, we calculate the expected cost of acceptance as a function of the prior probabilities and decision costs. If we take another observation, then a dead mutant leads to accepting the sample, incurring a decision cost (W_{12}), and a live mutant will take us to a point where we either continue the test or reject immediately, incurring a decision cost (W_{21}). We determine λ_1 to be that value for which these two costs of accepting immediately and of taking one more observation to continue the test, if necessary, are equal. It is noted, in general, that the difference between the expected cost of an immediate decision and expected cost of continuing the test should decrease as the sample (mutant) size increases considerably.

As a result of the optimization procedure, [18] i.e., equating the expected cost of an immediate decision to the expected cost of at least one more mutant inspected, we have the following equations as a result:

For $\rho = (\lambda_2/\lambda_1) < (p_1/p_2)$,

$$\lambda_1 = \frac{W_{12}p_2R(p_2/p_1, (\lambda_2p_2)/(\lambda_1p_1); p_2) - c - p_2cS(p_2/p_1, (\lambda_2p_2)/(\lambda_1p_1); p_2)}{W_{21}p_1R(p_2/p_1, (\lambda_2p_2)/(\lambda_1p_1); p_1) + c + p_1cS(p_2/p_1, (\lambda_2p_2)/(\lambda_1p_1); p_1)}; \tag{10}$$

$$\text{and for } \rho = \frac{\lambda_2}{\lambda_1} > \frac{p_1}{p_2},\ \lambda_1 = \frac{W_{12}p_2 - c}{W_{21} + c} \tag{11}$$

Similary,

$$\rho = \frac{\lambda_2}{\lambda_1} < \frac{q_1}{q_2},\ \lambda_1 = \frac{W_{12}q_2A(q_2/(\rho q_1), q_2/q_1; p_2) + c + q_2cS(q_2/(\rho q_1), q_2/q_1; p_2)}{W_{21}q_1A(q_2/(\rho q_1), q_2/q_1; q_1) - c - q_1cS(q_2/(\rho q_1), q_2/q_1; p_1)} \tag{12}$$

7 Barnard's Score Notation for a Binomial SPRT

The following derivation is from [18]:

According to Barnard [3], any sequential probability ratio test procedure for a binomial population can be reduced (without much loss in accuracy) to a scoring procedure as follows: Take the logarithms on both sides of (9) and divide throughout by $\log(q_1/q_2)$:

$$\frac{\log L}{\log(q_1 q_2)} < X - Y\frac{\log(p_2/p_1)}{\log(q_1/q_2)} < \frac{\log U}{log(q_1/q_2)} \tag{13}$$

and then define:

$$H_1 = \frac{\log U}{\log(q_1/q_2)} \tag{14}$$

$$H_2 = \frac{\log L}{\log(q_1/q_2)} \tag{15}$$

If we round off b, H_1 and H_2 to the nearest integers, the sequential likelihood ratio test procedure reduces to the following scoring scheme: Start with a score H_2, and add one to the score for each dead mutant observed, and subtract b for each live mutant found. Reject the sample (assume the test data is not strong enough) if the score falls to zero or less, and accept the sample if the score reaches $2H = H_1 + H_2$.

As a procedure for calculation, in practice, we shall first have to fix values for a_1, a_2, p_1, p_2, W_{12}, W_{21}, and c. The quantity b can then be calculated by

$$b = \frac{\log(p_2/p_1)}{\log(q_1/q_2)} \tag{16}$$

Then calculate

$$2H = \frac{-\log(\rho)}{\log(q_1/q_2)} \tag{17}$$

Where ρ can be estimated by calculating the interval given by:

$$\frac{c^2}{[w_{12}(q_1 - q_2) - c][W_{21}(q_1 - q_2) - c]} < \rho < 1 \tag{18}$$

From experience, it appears that the lower limit, or any number a little higher than the lower limit, can serve as a good first guess for ρ in order to start the iteration. Formulae for the average sample size and acceptance probability of such a scheme have been given by Burman [7] and Anscombe [1]. They are exact if b, H_1 and H_2 are integers. The error involved in rounding off is small if b is greater than 10, and this is often the case in practice. If we replace $A(U, L; p)$ etc., as defined earlier, by $A(H_1, H_2; p)$ etc., in score notation, we get, if $\rho < p_1/p_2$:

$$\rho = F(\rho) = \frac{W_{12}q_2A(2H-1,1;p_2) + c + q_2cS(2H-1,1;p_2)}{W_{21}q_1A(2H-1,1;p_1) - c - q_1cS(2H-1,1;p_2)} \tag{19}$$
$$* \frac{W_{21}p_1R(b,2H-b;p_1) + c + p_1cS(b,2H-b;p_1)}{W_{12}p_2R(b,2H-b;p_2) - c - p_2cS(b,2H-b;p_2)}$$

If $\rho > (p_1)/(p_2)$, we get

$$\rho = F_1(\rho) = \frac{[W_{12}q_2A(2H-1,1;p_2) + c + q_2cS(2H-1,1;p_2)][W_{21}p_1 + c]}{[W_{21}q_1A(2H-1,1;p_1) - c - q_1cS(2H-1,1;p_1)][W_{12}p_2 - c]} \tag{20}$$

In both cases, b and $2H$ are as denoted in (16) and (17).

To solve equations (19) and (20), we start with some guessed values for ρ and proceed with the iteration until we get the same value of ρ. Hence, when we get a value that gives rise to the same value of $2H$ as was used in the previous iteration, we stop and take that as our solution. Once ρ is obtained, we calculate λ_1 and λ_2 from equations (10) and (12) or (11) and (12).

Wald, Barnard and Bartlett [20, 3, 4] have given simple asymptotic formulae for the chance of acceptance and average sample size of an open scheme. Setting $X = p(b+1)$ we consider the limit $p \to 0$ with X constant. Wald's formula for the chance of acceptance then becomes

$$A = \frac{\exp(R_2T) - 1}{\exp(R_2T) - \exp(-R_1T)} \tag{21}$$

where

$$X = \frac{T}{(\exp(T) - 1)}, \; R_1 = \frac{H_1}{(b+1)}, \text{ and } R_2 = \frac{H_2}{(b+2)}$$

Barnard and Bartlett's formula lead to the same result except that X is replaced by $X = 2/(T+2)$, which is equivalent to the earlier X formula when X is near 1. Wald and Bartlett [4, 19] give a formula for the average sample size S as defined by:

$$\frac{S}{b+1} = \frac{R_1P - R_2(1-P)}{1-X} \tag{22}$$

provided $X \neq 1$, while for $X = 1$

$$\frac{S}{b+1} = R_1R_2 \tag{23}$$

as $p \to 0$, S and $(b+1) \to \infty$, and their ratio tends to the limit shown.

8 Illustrative Examples as Applied to Mutation

To illustrate this method, we applied it to the *trityp* program, first presented in [15]. This program takes as input three integers representing the length of the sides of a triangle, and then determines whether sides define scalene, isosceles, equilateral, or illegal. This program was chosen because of its widespread use in the testing literature, and because of our familiarity with it in other work (e.g., [13] and [16]).

The routine contains 30 lines of Fortran-77 code. The MOTHRA mutation environment generates 951 mutants for this program, 107 of which are equivalent (leaving 844 non-equivalent). The smallest test set we know of that can kill 100% of non-equivalent mutants has 27 test cases.

Our proposed most economical sequential test method is as follows:

1. Initialization. Set $Sum = H_2$. Set $\mathcal{T} = \emptyset$, where $\mathcal{T}$ is the set of test cases generated so far. Generate the non-equivalent mutants, $\mathcal{M}$, for the program. Set $\mathcal{H} = \emptyset$, where $\mathcal{H}$ is a holding set for mutants. Go to step 2.

2. Select a mutant $m \in \mathcal{M}$ at random. $\mathcal{M} = \mathcal{M} - m$. Go to step 3.

3. Execute m against $t, \forall t \in \mathcal{T}$. If m is killed, remove it from $\mathcal{M}$ and go to step 4, else $\mathcal{H} \leftarrow \mathcal{H}+m$ and go to step 5.
4. $Sum = Sum + 1$ If $Sum >= 2H$ accept the test set and stop, else go to step 2.
5. $Sum = Sum - b$. If $Sum > 0$ go to step 2, else go to step 6.
6. Develop a new test case, τ that kills the current mutant, m. Set $\mathcal{T} \leftarrow \mathcal{T} + \tau$. Go to step 7.
7. $\forall m \in \mathcal{H}$ execute m against τ. If m is killed, set $\mathcal{H} \leftarrow \mathcal{H} - m$, set $Sum = Sum + b + 1$. If $Sum >= 2H$, accept the test set and stop. Otherwise, go to step 2.

In short, we are adding test cases to $\mathcal{T}$ until we are able to kill a total of H_1 mutants, selected at random, when executed against the cases in $\mathcal{T}$. After this is accomplished, we stop, and then use $\mathcal{T}$ to test the original, non-mutated code.

For our tests, we used the prior probabilities and costs developed for the examples in [18]. The following tables illustrate our results. The first column in each table indicates the number of mutants selected in step 2 before it was necessary to generate a new testcase. The second column indicates the number of live mutants after executing **all** mutants against the test cases accumulated in $\mathcal{T}$ to that point.

In each of the following tables, the first column represents the number of test cases that are in the set $\mathcal{T}$ at that point in the test. The second column represents the number of mutants executed (as in step 3) before we stop to accept the test set or generate a new test case to add to $\mathcal{T}$. The third column indicates the number of non-equivalent mutants that would still be considered live if they **all** were executed against the test set $\mathcal{T}$, as would be done in the deterministic method of mutation testing.

8.1 Example 1

In this example,

$$a_1 = \frac{5}{9}, a_2 = \frac{4}{9}, p_1 = 0.01, p_2 = 0.10, W_{21} = 400, W_{12} = 500, \ \textit{and}\ c = 1$$

The necessary calculations result in values of

$$H_2 = 34, 2H = H_1 + H_2 = 68, b = 24$$

Thus, the test set was accepted after generating 14 test cases when Sum, as described in section 8, reached $2H = 68$, or exceeded it. These 14 test cases form a test set that kills over 93% of the live mutants using the deterministic approach, and that is acceptable at the level of 99% (i.e., $100(1 - p_1)\%$) with our proposed testing approach.

Note that in the worst case, the number of mutant executions required is given by:

$$0 * 1 + 1 * 3 + 2 * 5 + 3 * 6 + \ldots + 13 * 16 + 1 = 1116$$

as compared to the deterministic case requiring

$$0 * 844 + 1 * 692 + \ldots + 13 * 61 = 5361$$

8.2 Example 2

In this example,

$$a_1 = 0.5981, a_2 = 0.4019, p_1 = 0.008663, p_2 = 0.041856, W_{21} = 143.6, W_{12} = 266, c = 1$$

The necessary calculations result in values of

$$H_2 = 12,\ 2H = H_1 + H_2 = 44,\ \text{and}\ b = 46$$

In this case, we accept the test set after generating 11 test cases that kill 88.4% of the live mutants using the deterministic approach, and is acceptable at the level of 99.13% with our suggested approach.

9 Summary Discussion and Conclusions

As Barnard [3] has remarked, Wald's sequential procedure for testing the fraction defective of a batch of articles can be conveniently expressed in terms of a scoring system, as follows: [18]

> Sample the batch by drawing articles randomly from it one by one. Add 1 for a non-defective article, $-b$ for a defective. With starting score zero, accept the batch if the score reaches or exceeds H_1, reject it if the score falls to or below H_2. Continue sampling until one or another decision is reached. Or, identically, sample the batch, count $+1$ for a non-defective item, $-b$ for a defective. With starting score H_2, accept the batch if the score reaches $2H$, reject it if the score falls to zero or less.

In our mutation-oriented testing, the acceptance and rejection is of a test data set necessary to thoroughly test a piece of software. Once we accept a test set, we execute the original program against it to see if the software is correct. If we reject a test data set, we continue to augment the test data set until it is accepted.

When compared with the traditional method of mutation testing, this approach results in a great reduction in the number of executions required to judge a test set as adequate. Using the deterministic method may result in literally millions of executions as each mutant is executed against each test case, as the number of mutants is on order of the number of lines in the program, squared. However, our method may result in only a few hundred or thousand mutant executions to achieve a test set that can be used to test the software to the same level of confidence.

For instance, in our first example, the statistical sequential method we have described required no more than 1116 mutant executions, where the deterministic method required 5361 executions. Furthermore, our proposed SPRT resulted in a test data set with a statistically-determined probability of 99% adequacy, while the deterministic method resulted in a measure of only 93% of the mutants killed. Similarly, in example 2, our method required no more than 218 mutant executions compared to the 4196 executions required on the same test data for the deterministic case. Our method provided a test set 99.13% accurate (assuming correct priors and cost factors), while the deterministic method applied to the same test set only provides a measure of 88.4% adequacy.

We believe that the efficiency of our sequential approach, coupled with power of mutation testing, provides a cost-effective form of generating reliable test sets. Our proposed statistical sequential method requires only that we be able to establish costs of faulty acceptance, rejection, and sampling (mutant selection and test case generation), and prior testing probabilities—all of which are possible to do in most production environments.

References

[1] F. J. Anscombe. Tables of sequential inspection schemes to control fraction defective. *Journal of the Royal Statistical Society*, CXII(Part II):180–206, 1949.

[2] S. Bai and Y. Yun. Optimum number of errors corrected before releasing a software system. *IEEE Transactions on Reliability*, 37(1):41–44, April 1988.

[3] G. A. Barnard. Sequential tests in industrial statistics. *Journal of the Royal Statistical Society*, Suppl. 8:1–27, 1946.

[4] M. S. Bartlett. The large-sample theory of sequential tests. *Proceedings of the Cambridge Philosophy Society*, 42, 1946.

[5] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[6] C. Bullard and E. H. Spafford. Testing experience with Mothra. In *Proceedings of 25th Southeast ACM Conference*, Birmingham AL, April 1987.

[7] J. P. Burman. Sequential sampling formulae for a binomial population. *Journal of the Royal Statistical Society*, Suppl. 8:98–103, 1946.

[8] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra tools set. In *Proceedings of the 22nd Hawaii International Conference on Systems and Software*, pages 275–284, Kona, Hawaii, January 1989.

[9] R. A. DeMillo and A. J. Offutt. Experimental results of automatically generated adequate test sets. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, Portland, Oregon, September 1988.

[10] Richard A. DeMillo and Eugene H. Spafford. The Mothra software testing environment. In *11th Nasa Software Engineering Laboratory Workshop*. Goddard Space Center, December 1986.

[11] R. Govindarajalu. *Sequential Statistical Procedures*. Academic Press, 1975.

[12] W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, 1987.

[13] R. J. Martin. Using the Mothra software testing environment to ensure software quality. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, Dayton, OH, May 1989.

[14] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, NY, 1979.

[15] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[16] Eugene H. Spafford. Extending mutation testing to find environmental bugs. *Software Practice and Experience*, 20(2):181–189, February 1990.

[17] Statistics Research Group. *Sequential Analysis of Statistical Data: Applications.* Columbia University Press, 1945.

[18] M. K. Vagholkar and G. B. Wetherill. The most economical binomial sequential probability ratio test. *Biometrika*, 47(1 and 2):103–109, 1960.

[19] A. Wald and J. Wolfowitz. Optimim character of the sequential probability ratio test. *Annals of Mathematical Statistics*, 19:326–339, 1948.

[20] Abraham Wald. *Selected Papers in Statistics and Probability.* McGraw Hill, 1945.

[21] P. J. Walsh. *A Measure of Test Case Completeness.* PhD thesis, Watson School of Engineering, State University of New York at Binghamton, Binghamton, NY, 1985.

[22] G. B. Wetherill and K. D. Glazebrook. *Sequential Methods in Statistics.* Chapman and Hall, 1986. Third edition.

[23] Lee J. White and E. K. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[24] M. R. Woodward and K. Halewood. From weak to strong: Dead or alive? an analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 152–158, Banff, Alberta, Canada, July 1988.

Cases in $\mathcal{T}$	Mutants Examined	Total Live Mutants
0	1	844
1	3	692
2	5	627
3	6	538
4	7	510
5	8	453
6	11	433
7	27	345
8	32	282
9	35	199
10	60	106
11	90	93
12	108	71
13	116	61
14	accept	53

Cases in $\mathcal{T}$	Mutants Examined	Total Live Mutants
0	1	844
1	2	692
2	3	575
3	5	538
4	6	451
5	7	423
6	8	369
7	20	351
8	24	337
9	27	243
10	32	217
11	accept	98