

The Revealing Power of a Test Case*

Jeffrey M. Voas[†]

Keith W. Miller[‡]

Abstract

“Propagation, infection, and execution analysis” (termed *PIE*) is used for predicting where faults can more easily hide in software. To make such predictions, programs are dynamically executed with test cases. In this paper, we collect information concerning those test cases into a histogram. Each bin in a histogram represents a single test case. The score in a bin predicts the likelihood that the test case will reveal a fault through the production of a failure (if a fault exists in the set of program locations that the test case executes). Preliminary experiments using program mutations suggest that the histogram technique presented in this paper can rank test cases according to their fault revealing ability.

Index Terms

“Propagation, Infection, and Execution Analysis (*PIE*),” histogram, mutation testing, coverage criteria, fault revealing ability, probability, software testing.

1 Introduction

Software testing is a ubiquitous technique, but many of its characteristics are still poorly understood. To approach testing scientifically, we must be able to quantify test quality. However, measurement of tests and testing have tended to be qualitative and collective, assigning approval or a percentage approval to a group of test cases. In this paper we propose an empirical technique that produces a numerical value (or score) representing the ability of a single test to reveal the existence of a fault.

When a test case is executed and the resulting output is compared with the expected output, two kinds of information can be gained. (We assume that an oracle exists; i.e., the “expected” output is the *correct* output.) If the actual output matches the expected output, we are certain that the program works for this particular output. If the actual output does **not** match the expected output, we say that a *failure* has occurred, and we know that a fault exists.

*Supported by a National Research Council NASA-Langley Resident Research Associateship and NASA-Langley Grant NAG-1-884.

[†]Reliable Software Technologies Corporation, Penthouse Suite, 1001 N. Highland Street, Arlington, VA 22201.

[‡]Department of Computer Science, College of William & Mary, Williamsburg, VA 23185.

We define the *fault revealing ability* of a test case to be a measure of the likelihood that the test case will produce a failure if a fault were to exist anywhere in the program. A test case with high fault revealing ability has an obvious and immediate benefit for software development. However, the import of a test case that does **not** reveal the existence of a fault is more problematic. Unless exhaustive testing occurs, we cannot be sure that a subset of the program’s domain is representative of the fault revealing ability of all possible test cases. Much of testing research creates hueristics that suggest a test suite that will **probably** catch **most** faults, and whose success, therefore, suggests software with a low probability of failure.

For example, various test coverage criteria codify the idea that a suite of tests should exercise the code in certain ways or with a certain intensity in order to uncover a maximum number of faults in a minimum amount of testing. Statement coverage and limited path coverages are two widely used techniques.

Mutation testing is another method for designating a suite of test cases as “good.” Mutation testing generates syntactic variations of the program being tested. A suite of tests is good if it successfully distinguished the semantically distinct mutants from the original with at least one failure for each mutant.

To examine the efficacy of these approaches, consider that a test case must do the following three things in order to reveal the existence of a fault:

1. executes the code where the fault has an impact (“execution”),
2. changes the data state of the computation (“infection”), and
3. propagates this erroneous state to an output (“propagation”).

This three part model, first described by [12] and later by [17], can be used to identify fundamental differences between coverage criteria and mutation testing criteria. Coverage criteria focus on the first part of the model, determining how the execution proceeds through the code. Statement coverage, branch coverage, and path coverage criteria all ensure that a set of test cases exercise some portion of the code; these are structure-based criteria. Data flow coverage criteria [16, 15] differ from structure-based criteria by looking at the flow of data instead of the flow of control. Data flow coverage criteria produce test cases that attempt to ensure that the result of a computation is used by subsequent computations [16, 15]. Structural coverage criteria do not reflect information about whether a fault would affect the data state with these test cases, or whether the effect would be likely to propagate to output. Data flow coverage criteria can reflect information in the test cases it selects concerning whether all locations were executed. Data flow coverage criteria can also reflect information concerning whether a fault would affect the data state if the *use* is incorrect, and whether the effect would be likely to propagate to output if the *def* is wrong [11]. Since data flow only reflects propagation information in this special case, data flow analysis is a weak approximation, in general, to the three-part failure model. However, data flow analysis is a much less costly analysis than *PIE*.

Strong mutation testing [1] **does** embody all three aspects of the three part model, but it includes only those test cases that exhibit all three characteristics on at least one of the mutant programs. Weak mutation testing [2] and firm mutation testing [23] both require less from the test suite: in weak mutation testing, each mutant must only execute and create

an altered data state; in firm mutation testing, each mutant must execute, create an altered data state, and propagate an altered data state beyond the point where it is created, but not necessarily to the output.

In this paper, we present an alternative method for evaluating test cases. In this method, we quantify the fault revealing ability of each test case by measuring empirically the test’s capacity for executing locations, infecting if a fault exists, and propagating infections. Based on the three part model of software failure [12, 17], the method can be used both on individual test cases and on a group of test cases.

The rest of this paper proceeds in five parts. In Sections 2 and 3, we extract a description of the technique “propagation, infection, and execution analysis (*PIE*)” from [22]. *PIE* is a technique for quantifying the interaction between source code and test data.

§4 presents “*PIE* histograms,” a method by which we can assign a numeric score to an individual test case, augmenting the *PIE* technique to collect additional information that will allow us to predict the fault revealing ability of an individual test case. Once *PIE* has been used to assign histogram scores to individual test cases, we may want to use this information to build test suites similar to those assembled in other techniques; however, *PIE* gives more information about which individual tests will tend to be most effective within a suite. The paper concludes with details of preliminary experiments that suggest this approach has merit.

2 The Theoretical *PIE* Model

PIE is a technique based on the three part model of software failure [12, 17]. *PIE* is composed of three subtechniques: propagation analysis, infection analysis, and execution analysis. Elsewhere, we have described using these measurements in design, debugging, and in software quality measurement [19, 22]. In this paper, we focus on *PIE*’s utility in predicting the fault revealing ability of test cases.

2.1 General Definitions

We view a program as an implementation of a function g , that maps a domain of possible test cases to a range of possible outputs. Another function, f , with the same domain and perhaps different range, represents the desired behavior of g , and can be thought of as a functional specification of g . An *oracle* is a recursive predicate on input-output pairs that checks whether or not f has been implemented for an input, i.e., oracle $\omega(x, y)$ is TRUE iff $f(x) = y$. Then the oracle is used with $g(x)$ for y . During testing, it is necessary to be able to say whether a particular output of g is *correct* or *incorrect* for a particular test case x , with the latter implying that $g(x) \neq f(x)$, and the former implying that $g(x) = f(x)$. The *failure probability* of program P , with respect to an test case distribution D , τ_{PD} , is the probability that P produces an incorrect output for a test case selected at random according to D .

In the technique, it is necessary to uniquely identify specific syntactic program constructs as well as the internal data states created during execution. To uniquely identify syntactic constructs, we define a *location* to be either an assignment statement, an input statement,

an output statement, or the <condition> part of a **if** or **while** statement. Our definition for location is based on Korel’s [4] definition for a single instruction.

A program *data state* is a set of mappings between all variables (declared and dynamically allocated) and their values at a point during execution; in a data state we include both the program test case used for this execution and the value of the program counter. A data state is only observed between two dynamically consecutive locations. The execution of a location is considered here to be atomic, hence data states can only be viewed between locations. As an example, the data state

$$\{(\text{test case}, 3), (\mathbf{a}, 5), (\mathbf{b}, 5), (\mathbf{c}, \text{undefined}), (\text{pc}, 10)\}$$

tells that variables **a** and **b** have the value 5, the next instruction executed is at address 10, variable **c** is undefined, and the program test case that started this execution was a 3. Before program execution on a test case begins, all variables are undefined.

A *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion between locations. We refer to a data state error as an *infection*, and use these two terms interchangeably. If a data state error exists, the data state and variable with the incorrect value at that point are termed *infected*. A data state may have more than one infected variable. *Propagation* of a data state error occurs when a data state error affects the output. *Cancellation* has occurred when the existence of a data state error is not discernible in the program output, i.e., after viewing the output, we have no indication that a data state error ever occurred. In this model, we do not look at intermediate locations for data state error cancellation; only at output locations.

If there exists at least one test case from a distribution D for which a program P fails, then we say that P contains a *fault* with respect to D . Even though we may know that a fault exists in a program, we cannot in general identify a single location as the exclusive cause of the failure. For example, several locations may interact to cause the failure, or the program can be missing a required computation which could be inserted in many different places to correct the problem. However, if a program is annotated with assertions about the correct data state before and after a particular location l , and if there exists a test case from D such that l ’s succeeding data state violates the assertion and l ’s preceding data state does not violate the assertion, then l contains a fault.

In the technique based on this model, it is important to be able to determine whether a particular variable at some specific location of a program has any potential impact on the output computation of the program. A variable is termed *live* at a particular location if this potential exists. Determination of whether a variable is live is made statically from a flow graph that is augmented with def-use information. Admittedly, certain variables defined as live via static analysis using a flow graph containing def-use information might not be defined as live if our definition were based on the dynamic behavior of the program [3].

2.2 Model Definitions

This section formalizes:

1. the probability that a location is executed—an execution probability.

2. the probability that a change to the source program causes a change in the resulting internal computational state—an infection probability.
3. the probability that a forced change in an internal computational state propagates and causes a change in the program’s output—a propagation probability.

To define execution, infection, and propagation probabilities, we first introduce notation. Recall this technique tracks data states as a program executes. It is therefore necessary to uniquely identify a data state according to the test case that the program is currently executing on, the location in the program where we are observing the data state, and which iteration of the location we are observing this data state on (if the location is executed more than once for this test case).

Let S denote a specification, P denote an implementation of S , x denote a program test case, Δ denote the set of all possible test cases to P , D denote the probability distribution of Δ , l denote a program location in P , and let i denote a particular execution (or iteration) of location l caused by test case x . Hence if $i > 1$, then l is in a loop or in a procedure that is called more than once. Let \mathcal{B}_{lPx} represent the data state that exists prior to executing location l on the i^{th} execution from test case $x \in \Delta$, and let \mathcal{A}_{lPx} represent the data state produced after executing location l on the i^{th} execution from test case x .

It is important for us to be able to group data states into sets with similar properties. For instance, assume that location l is executed n_{xl} times by input x . Then we might want to look at all of the data states that are created by this input immediately before l is executed or immediately after l is executed. The following sets allow us to do so:

$$\mathcal{B}_{lPx} = \{\mathcal{B}_{lPx} \mid 1 \leq i \leq n_{xl}\}$$

$$\mathcal{A}_{lPx} = \{\mathcal{A}_{lPx} \mid 1 \leq i \leq n_{xl}\}$$

We further group these sets for all $x \in \Delta$:

$$\beta_{lP\Delta} = \{\mathcal{B}_{lPx} \mid x \in \Delta\}$$

$$\alpha_{lP\Delta} = \{\mathcal{A}_{lPx} \mid x \in \Delta\}$$

We let f_l denote the function that is computed at a location l . The input to a function computed at a location is a data state and the output of such a function is also a data state. Thus

$$\mathcal{B}_{lPx} \xrightarrow{f_l} \mathcal{A}_{lPx}.$$

The *execution probability* ε_{lPD} of location l of program P is simply the probability that a randomly selected input x selected according to D will execute location l .

Let \mathcal{M}_l represent a set of z_l mutants of location l : $\{m_{l1}, m_{l2}, \dots, m_{lz_l}\}$ (where $1 \leq y \leq z_l$) [1, 2, 13, 14]. And let $f_{m_{ly}}$ denote the function computed by mutant m_{ly} . The *infection probability* $\lambda_{m_{ly}lPD}$ of mutant m_{ly} is the probability that the succeeding data state of location l is different than the succeeding data state that mutant m_{ly} creates, given that l and m_{ly} execute on a data state that would normally precede l (one that would be created by a randomly selected input according to D).

We define a *simulated infection* as a changed value forced into the value of some variable (that already had a value) in a data state. As we have already stated, \mathcal{A}_{lPx} denotes the

data state created after the i^{th} iteration of location l on input x ; $\check{\mathcal{A}}_{lPix}$ denotes this same data state after a simulated infection is injected into \mathcal{A}_{lPix} . A simulated infection affects a single live variable.

The *propagation probability* $\psi_{a|lPD}$ for a simulated infection affecting variable a in the i^{th} data state succeeding location l (where this data state is created by a randomly selected input x according to D) is the probability that P 's output differs (from what would normally be produced) after execution is resumed using the simulated infection.

3 Approximating the Theoretical Model

We can estimate the previous three conditional probabilities using a class of test cases that are selected at random according to D . The three previous conditional probabilities are defined for a single randomly selected test case; to find estimates of these probabilities, we will use a class of test cases.

Three analyses are used for estimating the execution probability, infection probability, and propagation probability: Execution analysis, Infection analysis, and Propagation analysis. These methods are the focus of §3 and collectively are termed *PIE* analysis.

Before we perform these analyses, we assume several properties about the state of the program and our knowledge about its environment:

1. The program is close to being correct, meaning that it compiles and is believed to be close to a correct version of the specification both semantically and syntactically; this essentially is the *competent programmer hypothesis* [1]. This closeness is useful because the confidence in the applicability of the resulting estimates diminishes as we move further away from the assumption.
2. A distribution of test cases, D , is available from which we can sample.
3. The test cases that we sample are only from Δ .
4. The cardinality of Δ is effectively infinite for sampling purposes. Although there are finitely many numbers representable on a computer, we will assume this fixed number exceeds what can be exhaustively sampled from during testing in a practical amount of time.

3.1 Execution Analysis

Execution analysis is related to other analyses that are based on program structure. Structural testing methods attempt to cover specific types of software structure with at least one test case. For example, *statement testing* is a structural testing method that attempts to execute every statement at least once; *branch testing* is a structural testing method that attempts to execute each branch at least once. *Execution analysis* estimates the probability of executing a particular location when test cases are selected according to a particular test case distribution.

Statement testing and branch testing are weak criteria because their satisfaction does not ensure failure should a fault exist. Executing a statement during statement testing and

not observing program failure merely provides *one* data point for estimating whether or not the statement contains a fault. Execution analysis benefits structural testing methods by indicating the likelihood of executing a particular statement.

Execution Analysis estimates execution probabilities. The *execution estimate* of execution probability ε_{lPD} is denoted by $\hat{\varepsilon}_{lPD}$ —it is found by finding the proportion of inputs (selected according to D) that cause location l is executed. A potential exists for inputs that are selected according to D to appear to cause non-terminating computations. For this reason, if a non-terminating computation is suspected, a mechanism within execution analysis will be required that will terminate execution analysis on that input (meaning that input will be ignored). This situation may cause the resulting execution estimates to be a function of some input distribution other than D , but regardless, such a mechanism is needed.

3.2 Infection Analysis

Infection analysis is similar to the processes employed in fault-based testing. *Fault-based testing* aims at demonstrating that certain faults are not in a program [9, 8, 10, 12, 7, 17, 24]. Morell [12, 9, 8, 10, 7] proves properties about fault-based strategies concerning certain faults that can and cannot be eliminated using fault-based testing. Since fault-based testing restricts the class of possible faults, the possible testing is limited. Fault-based testing defines faults in terms of their syntax.

Fault-based testing also evaluates test cases based on their ability to distinguish the specific faults. *Mutation testing* [1, 2, 13, 14] is a fault-based testing strategy that does just this—it evaluates program test cases. It takes a program P and produces n versions (*mutants*) of P , $[p_1, p_2, \dots, p_n]$, that are syntactically different from P . The goal of *strong mutation testing* [1] is to find a set of test cases that distinguishes the mutants from P .

Another type of mutation testing, *weak mutation testing* [2], selects test cases that cause all imagined infections to be created by a possibly infinite set of mutants. Infection analysis differs from weak mutation testing in that infection analysis measures the effect mutants have on succeeding data states. Syntactic changes are made to program locations and infection analysis finds the probability that a particular mutant affects the data state. In other words, infection analysis tests a location’s ability to sustain a syntactic change yet not alter the data state that results from executing the mutant. When a syntactic mutant alters the data state, we call this data state an *altered data state*. In short, weak mutation testing and infection analysis are distinct.

Infection analysis estimates an infection probability for each $m_{ly} \in \mathcal{M}_l$. The estimate of some infection probability, $\lambda_{m_{ly}lPD}$, is termed an *infection estimate*, and is denoted by $\hat{\lambda}_{m_{ly}lPD}$ —it is found by the following algorithm:

1. Set variable **count** to 0.
2. Randomly select test cases according to D until we find a test case x upon which P halts in a fixed amount of time.
3. Find the corresponding \mathcal{B}_{lPx} in $\beta_{lP\Delta}$. Uniformly select a data state, \mathcal{Z} , from \mathcal{B}_{lPx} .
4. Present the original location l and the mutant m_{ly} with data state \mathcal{Z} and execute both locations in parallel.

5. Compare the resulting data states and increment **count** when $f_l(\mathcal{Z}) \neq f_{m_{ly}}(\mathcal{Z})$.
6. Repeat steps 2-5 n times.
7. Divide **count** by n yielding the sample mean of $\frac{\text{number of times that } f_l(\mathcal{Z}) \neq f_{m_{ly}}(\mathcal{Z})}{n}$; this is our $\hat{\lambda}_{m_{ly}lPD}$.

The mutants that have been used in this research have been limited to mutants of arithmetic expressions and predicates. For arithmetic expressions, the mutants considered in our research are limited to single changes to a location—this is similar to the types of mutations made in mutation testing. Our assignment statement mutants are: (1) a wrong variable substitution, (2) a variable substituted for a constant, (3) a constant substituted for a variable, (4) expression omission, and (5) a wrong operator. For boolean predicates, the mutants are: (1) substituting a wrong variable, (2) exchanging **and** and **or**, and (3) substituting a wrong equality/inequality operator. We have purposely limited the syntactic changes to single changes to avoid the explosion that occurs in the number of combinatorial changes that could be made at each location.

All of the problems associated with generating mutants that are not semantically equivalent in mutation testing exist here as well. In this initial stage of our research, we have closely followed the mutation techniques developed by [1, 2, 13, 14]; as our experience with *PIE* increases, we expect to gain insight into the strengths and weaknesses of different code mutating techniques.

3.3 Propagation Analysis

In this section, we discuss using simulated infections to predict the propagation of actual infections (if they exist). Unless otherwise distinguished, the term “infection” in this section refers to a simulated infection.

Propagation analysis generalizes the concept of fault-based testing by analyzing classes of faults in terms of their semantic effect on a data state. Propagation analysis directly modifies program data states and measures this effect on the computation of the program. Mutation testing modifies the program text in search of test cases that kill mutants. It is only through the notion of mutating that propagation analysis and mutation testing are related. They greatly differ in the information they gather.

Propagation analysis estimates propagation probabilities. The *propagation estimate* of propagation probability $\psi_{a\check{a}lPD}$ is denoted by $\hat{\psi}_{a\check{a}lPD}$ —it is found by the following algorithm:

1. Set variable **count** to 0.
2. Randomly select test cases according to D until we find a test case x upon which P halts in a fixed amount of time.
3. Find the corresponding \mathcal{A}_{lPx} in $\alpha_{lP\Delta}$. Set \mathcal{Z} to \mathcal{A}_{lPx} .
4. Alter the sampled value of variable a found in \mathcal{Z} creating $\check{\mathcal{Z}}$ and execute the succeeding code on both $\check{\mathcal{Z}}$ and \mathcal{Z} . Possible methods for altering a are discussed below.

5. For each different result in program output after termination on \check{Z} and Z increment **count**; increment **count** if a time limit for termination has been exceeded. This precaution is necessary because of the effects that perturbed variables can cause to the condition that terminates indefinite loops. In general, we cannot be certain that termination will not eventually occur, however we must set some time limit, or the analysis might never terminate. Also, we cannot be certain that the computation on Z will terminate; [22] presents insights into the ramifications of this problem.
6. Repeat steps 2-4 n times.
7. Divide **count** by n yielding the sample mean of $\frac{\text{number of times that program output differed}}{n}$; this is our $\hat{\psi}_{aiPD}$.

3.4 Understanding the Resulting Estimates

When *PIE* is completed for the entire program, we have three sets of probability estimates for each program location l in P given a particular distribution D :

1. Set 1: execution estimate—the estimate of the probability that program location l is executed.
2. Set 2: infection estimates—the estimates of the probabilities, one estimate for each mutant in \mathcal{M}_l at program location l , that given the program location is executed, the mutant will adversely affect the data state.
3. Set 3: propagation estimates—the estimates of the probabilities, one estimate for each live variable in (a_1, a_2, \dots) at program location l , that given that the variable in the data state following location l changes, the program output that results changes.

Note that each probability estimate has an associated confidence interval, given a particular level of confidence and the value of n used in the algorithms. The computational resources available when *PIE* is performed will determine an n for each algorithm.

3.5 Computational Costs of *PIE*

The costs of performing *PIE* fall into two categories. First, there is the cost of instrumenting the program with enough additional code to:

1. reveal when a location is executed,
2. create the syntactic mutations,
3. inject the simulated infections, and
4. perform bookkeeping duties that gather the necessary information.

The second cost category is the computational resources to perform the analysis. For programs that are greater than a thousand lines in length, these costs becomes large.

To date, no fully automated system is available to reduce the manual effort necessary to produce a complete *PIE* analysis. Because of this, experimentation with *PIE* has only been applied completely to small programs, or applied partially to large programs. *PIE* is currently being converted into a commercial system; when this system is operational, it will facilitate large scale experimentation.

We have applied propagation and execution analysis to selected locations in the 2000 line battle simulation described in [18] with considerable success [22]: we have found that our approximation to the model appears to be accurate enough to predict the effect that actual faults and actual data state errors create in programs. We have not yet applied infection analysis to this project. This type of success of propagation and execution analysis has been shown both for programs of this size (2000 lines in length) as for several programs of 10 lines or less.

PIE produces estimates that are as accurate for larger programs as for smaller programs. However the computational cost of performing *PIE* is roughly quadratic in the number of locations that are executed. For instance, if the computational cost of performing *PIE* on a 1000 line program is 1,000,000 units, then the computational cost of performing *PIE* on a 10,000 line program is on the order of 100,000,000 units. The cost of instrumenting the program to perform the analysis also increases substantially as the size of the program increases. Generally as the size of a program increases, the size of the data states increases, making the sets $\alpha_{IP\Delta}$ and $\beta_{IP\Delta}$ impossible to store in practice. The resulting execution strategies are again computationally expensive. None of these costs are disabling in a fully automated system, but these costs make extensive hand experimentation prohibitively time consuming.

4 Estimating the Fault Revealing Ability of Test Cases

In the previous section, we described how propagation estimates, infection estimates, and execution estimates are found using random test cases. Together, these estimates characterize the behavior of source code and the test case distribution from which its test cases are drawn. The application of *PIE* to predicting where faults can more easily hide from random tests is described in [20]. In this paper, we discuss how, for a given test case and a given program, *PIE* can be used to form a prediction of whether or not the test case will tend to reveal software faults. §4 describes two different histogram methods for applying *PIE*'s algorithms to predict whether or not a test case will reveal software faults, and we argue that a histogram approach is a viable option for quantifying the fault revealing ability of a single test case.

4.1 Method I

Although *PIE* currently is designed to produce propagation estimates, infection estimates, and execution estimates, *PIE* can be augmented to reveal the following information about each test case it uses:

1. How many locations did the test case execute during execution analysis?
2. How many data state infections were produced from the syntactic mutants during infection analysis at the locations it executed?
3. How many simulated infections at the locations executed by this test changed the output during propagation analysis?

Certain test cases execute more locations, create more altered data states, and propagate simulated infections more frequently than others. These are test cases that we expect will uncover faults more rapidly than test cases that execute fewer locations, create fewer altered data states, and propagate simulated infections less frequently. By distinguishing more “active” test cases from less active test cases, we should be able to uncover more faults with fewer test cases.

To change this intuitive notion into a number for each test case, we define a histogram H . H is created to identify those test cases that tend to execute more locations, create more altered data states, and propagate simulated infections more frequently. Let N_p represent the number of locations that test case p executes. For each location j executed by p , there are S_j syntactic mutants used during infection analysis and there are R_j simulated infections used during propagation analysis. Let s_j represent the number of syntactic mutants that actually caused altered data states at location j with test case p , and let r_j represent the number of simulated infections that propagate from location j to the output with test case p . We then calculate two scores for test case p :

$$\alpha_p = \frac{1}{N_p} \sum_j \frac{s_j}{S_j} \quad (1)$$

$$\beta_p = \frac{1}{N_p} \sum_j \frac{r_j}{R_j} \quad (2)$$

In both equations, j iterates over all the locations executed by the test case p . Thus, α_p and β_p characterize the ability to create altered data states and propagate simulated infections of the test case p , averaged over all locations executed by p . The final equation for p 's histogram score combines three terms, one for the number of locations executed, one for the ability to create altered data states, and one for the ability to propagate simulated infections:

$$H_p = N_p \alpha_p \beta_p \quad (3)$$

As expected, a high α_p , a high β_p , and a large number of executed locations will produce a high histogram score. If two test cases are similarly successful in creating altered data states and propagating simulated infections at the locations they execute, then the test case that executes more locations has the higher score.

4.2 Method II

A potential difficulty with Method I is that the predominant parameter in equation 3 is N_p . If it happens that some test case p executes only a few locations because these locations

are purposely intended to rarely be executed (for example, in a range checking operation), then we might be fooled into believing that p has little fault revealing ability, when actually p has enormous fault revealing ability but only at the few locations that it executes. It is obviously true that locations that are rarely executed are places where faults can more easily hide during testing. Therefore the test cases that execute these locations have a special importance, an importance that H ignores. To make PIE more sensitive to this situation, we create a second histogram, H' , that removes the emphasis on the number of locations that a test case executes. This removes the bias against test cases that execute few locations. Note, however, that the H' measure still involves the execution part of the failure model, since the averaging required for the infection and propagation measures required extensive information about execution. The score of a bin in this “modified” histogram that represents test case p is given in equation 4.

$$H'_p = \alpha_p \beta_p \tag{4}$$

where α_p and β_p are defined as above.

Notice that in H and H' we have purposely avoided the notion of having a test suite, i.e., grouping test cases into a single bin, and have instead produced histograms that predict the fault revealing ability of a single test case. This allows for the most detailed information possible for a set of test cases because each member is individually evaluated.

4.3 Algorithm Analysis of the Histogram Technique

During execution analysis the program is run once and N_p is found. During propagation analysis there are R_p perturbed live variables at the N_p locations, so the program is executed R_p times. And during infection analysis there are S_p syntactic mutants injected. Since infection analysis is solely concerned with the data state resulting from the syntactic mutant and is not concerned with what happens at any subsequent locations in the code, the mean number of program executions performed during infection analysis per test case per location is $1/2$, so the mean number of program executions performed for the S_p syntactic mutants is $S_p/2$. The reason for dividing by 2 is that on average, to execute any program location by starting execution of the program from the first location, we will need to only execute $1/2$ of the program locations until we reach the location. Thus the average number of program executions required for finding the score for one test case is approximately

$$R_p + (S_p/2) + 1.$$

For a histogram of k test cases, the number of program executions required is approximately

$$k \cdot (R_p + (S_p/2) + 1).$$

Although computationally expensive it should be noted that PIE is highly amenable to parallel architectures as demonstrated in [21].

5 The *PIE* Histogram Technique and Other Testing Criteria

The most fundamental difference between using *PIE* to evaluate tests and other criteria is the basis of comparison: *PIE* is an empirical estimating technique based on the three part software failure model [12, 17]. Other testing criteria address either part of that model or address the whole model in a different way than *PIE* does. In §5, we explain how this histogram building technique compares to mutation testing and coverage criteria.

Generally testing criteria produce sets of test cases, and these criteria do not quantify the fault revealing value of individual test case members. This is the second important distinction between this technique and other criteria.

By having a quantified value for each test case, we can take a set of tests, find the sum of their histogram values, and produce an overall quantity that reflects of the fault revealing ability of the set. In certain circumstances, this may allow us to compare the fault revealing abilities of different sets of tests produced by different criteria. However, grouping histogram values produces complications that may make such comparisons problematic.

One particular problem in grouping histogram scores is that *PIE* histograms are not sensitive to the overlap in the statements executed by more than one test case. For example, assume that we are analyzing two sets of test cases, and that they have roughly the same *PIE* histogram scores. Assume further that all the tests in the first set execute precisely the same code, and that code is only a small fraction of the source code in the program. The second set executes all the source code. Unless the tester has special knowledge that faults are concentrated in the portion exercised by the first set, then the second set would be preferred. When two sets are equivalent on a coverage criteria, then *PIE* histograms can be used to favor one set or the other based on the fault revealing tendencies in each of the sets.

It may be that *PIE* histograms can be adapted to quantifying groups of test cases in a more sophisticated manner. Possible refinements to accommodate test sets has been deferred until more extensive experiments are completed on quantifying the fault revealing ability of single tests.

5.1 *PIE* Histogram Technique and Mutation Testing

Both mutation testing and infection analysis rely on the assumption that syntactic mutants (that are used in the techniques) will somehow be representative of actual faults. However, mutation testing relies more heavily on this assumption than the *PIE* histogram technique. In mutation testing, test cases are only accepted if they execute, affect the data state, and propagate this change for at least one mutant. In the *PIE* histogram technique, a test case can be accepted (or rejected) based on its quantified ability to execute, affect the data state, or propagate the change. Since the syntactic mutants used in both mutation testing and *PIE* can only approximate actual faults, *PIE*'s more precise modeling of the failure process has the potential for increased predictive power.

Consider the situation of two test cases y and z , both of which are chosen to be in a mutation adequate test suite. Assume that we also subject these two test cases to *PIE*

and find their respective histogram scores. Instead of merely saying that test cases y and z killed a certain number of mutants, the *PIE* histogram score predicts how test cases y and z compare in their ability to reveal both the existing syntactic mutants and unknown faults (which might cause the simulated infections used in *PIE*). This has two advantages:

1. By separating infection analysis (which is a function of a set of syntactic mutants) from propagation analysis (which is a function of a set of simulated infections), *PIE* quantifies a test case’s ability to expose altered data states regardless of any syntactic mutant that might create them. Since we cannot know what faults might occur, the function that creates simulated infections is based on a random distribution that simulates the impacts on data states of many fault classes. Propagation analysis is not tied into the implicit “syntactic mutant-based” propagation analysis of strong mutation testing.
2. A histogram quantifies the fault revealing ability of each test case, rather than a suite of test cases. Once tests are combined into a suite, we have no way of determining which test cases have a greater fault revealing ability.

In situations where traditional mutation testing is desired, the *PIE* histograms could be used to select test cases most likely to efficiently kill mutants. This could reduce the number of test cases necessary to create an adequate suite for a given set of mutants.

5.2 *PIE* Histogram Technique and Coverage Criteria

Although execution analysis is part of this technique, *PIE* does not enforce syntactic coverages. *PIE* and coverage criteria have orthogonal concerns: *PIE* helps select test cases likely to produce failures if faults exist, and coverage criteria help ensure that the code is adequately exercised so that the potential faults are executed. Since coverages have practical and intuitive appeal, it seems reasonable to combine *PIE* with coverage criteria to gain the advantages of each technique.

PIE can be augmented to include the bookkeeping required for coverage evaluations. Using this bookkeeping, a tester could build a test suite that satisfied a particular coverage criteria while selecting the coverage criteria tests from test cases with greater histogram scores. The strengths of both techniques are combined in the resulting test suites.

6 Preliminary Experiment

In order to demonstrate the usefulness of this histogram building technique, we have done preliminary experiments that compare it with strong mutation testing. In the one experiment reported here, the small program shown in Figure 1 was the object of our analysis.

We first explain how we performed *PIE* on this program in order to give the reader a better understanding of how *PIE* is applied. For each replication of the experiment, ten thousand random test cases (each a trio of integers) were generated for variables \mathbf{a} , \mathbf{b} , and \mathbf{c} . All test cases were subjected to *PIE* and H and H' were created over all test cases. The details of this process follow.

```

read (a,b,c);
{1} if a <> 0 then begin
{2}   d := b*b - 4*a*c;
{3}   if d < 0 then
{4}     x := 0
      else
{5}     x := (-b + trunc(sqrt(d))) div (2*a)
      end
      else
{6}   x := - c div b;
write (x);

```

Figure 1: Program P .

First we perform execution analysis for the code above using a particular test case distribution. For a test case p drawn at random from that distribution, we keep track of how many of the six locations were executed; this is N_p .

Second, we perform infection analysis. The infection analysis was based on 42 semantically different syntactic mutants distributed among the 6 locations. For each test case p , we found which locations were executed, how many syntactic mutants were available at that location, and how many of those mutants infected when executed after test case p initialized the program. For example, assume test case p executes 4 locations, where infection analysis used 4, 5, 6, and 12 syntactic mutations respectively at those locations. Assume further that 3, 5, 3, and 6 of those mutations infect when execution begins with test case p . Then $\alpha_p = \frac{1}{4}(\frac{3}{4} + \frac{5}{5} + \frac{3}{6} + \frac{6}{12}) = \frac{11}{16}$.

Third, we perform propagation analysis. In this experiment, we perturb variables \mathbf{a} , \mathbf{b} , and \mathbf{c} at location 1, \mathbf{a} , \mathbf{b} , \mathbf{c} , and \mathbf{d} at location 2, \mathbf{a} , \mathbf{b} , \mathbf{c} , \mathbf{d} at location 3, \mathbf{a} , \mathbf{b} , \mathbf{c} , \mathbf{x} at location 4, \mathbf{a} , \mathbf{b} , \mathbf{c} , \mathbf{x} at location 5, and \mathbf{x} at location 6. Assume that the test case p executes 4 locations, 1, 2, 3, and 5; assume further that perturbations of variable a propagate for all locations, and that the only other variable that propagates when perturbed is variable \mathbf{x} at location 5. Then $\beta_p = \frac{1}{4}(\frac{1}{3} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}) = \frac{16}{48} = \frac{1}{3}$. Using this example, $H_p = 4(\frac{11}{16})(\frac{1}{3}) = \frac{44}{48} = \frac{11}{12}$ and $H'_p = \frac{1}{4}H_p = \frac{11}{48}$.

In the following experiment we use H_p only. After building H and H' , our experiment continued by using the mutation transforms described in [5] to produce 119 semantically significant mutations of the program. Our goal was to determine how many of the 119 mutants were distinguished from the original program by each of 10,000 test cases. We expected that test cases with the highest bin scores would tend to kill more mutants than test cases with the lowest bin scores.

The 119 mutations of the original program were not evenly distributed among the program locations. Furthermore, the small number of locations in the original program made conclusions about execution analysis suspect. For these reasons we compared a test case's H' score with its ability to kill the mutations it encountered, not with the absolute number of mutants it killed. That is, for each test case we recorded how many mutations occurred in the locations it executed. A test case's *kill ratio* was calculated as the ratio of mutants killed to mutants encountered.

We used 10,000 random tests cases and 99% confidence intervals. The expected value for the kill ratios of all test cases was 0.305 ± 0.004 . The expected value of the 1,000 test

cases with the lowest H' scores was 0.174 ± 0.0034 . The expected value of the 1,000 highest H' test case scores was 0.525 ± 0.009 . Several replications of 10,000 test cases each yielded similar results.

These results are only preliminary; the code in question is small and the number of mutants used in the infection analysis was also limited. However, experience with this limited experiment has already suggested future directions for refining the technique and its evaluation. Areas for future research include:

1. Exploring the relationship between H and H' , and evaluating different figures of merit based on the three estimates produced by *PIE* analysis (N_p , α_p , and β_p).
2. When the analysis is fully automated, more extensive syntactic mutation can be applied during infection analysis. This could improve the estimates of fault-detecting ability.
3. By expanding the variables affected by simulated infections, even better estimates of fault-detecting ability may be forthcoming.
4. The interplay between the set of mutants and the random test cases needs to be further explored. The number of mutants killed by test cases after they are ranked by *PIE* suggests that this factor is important.
5. The experiment uses program mutations to gauge the fault revealing ability of test cases. However, mutation testing rests on certain assumptions about the nature of software faults and the representation of all software faults by simple syntactic mutations. The experiment is open to the same criticisms as mutation testing on this point (see [5] for a fuller discussion). An important next step in evaluating the technique is to apply it to large programs with known faults.
6. The value of an individual test case to an overall testing effort may depend more on the *type* of mutants it kills rather than on the *number* of mutants it kills. For example, if a majority of test cases kills a mutant $m1$ and hardly any test cases kill a mutant $m2$, then a test case that kills $m2$ may be of great value in tracking down a difficult bug in a program. A more sophisticated approach to estimating the value of a test case may include weighting the histogram bin scores more heavily towards test cases that reveal mutants that are extremely difficult to kill, and thus are killed by very few test cases.

Despite the limitations of this small experiment, the preliminary results suggest that *PIE* does produce useful information concerning the input space for testing. More elaborate experiments using larger programs will be forthcoming when automation of a *PIE* system is complete.

7 Concluding Remarks

This paper has discussed the *PIE* histogram building technique and shown experimentally that the resulting information reflected the fault revealing ability of a test cases. When the

PIE techniques are automated, additional experimental evidence will provide more conclusive evidence about the costs and benefits of this method.

The *PIE* histogram technique has two intrinsic advantages over syntactic coverage or mutation criteria that create test suites:

1. Instead of grouping test cases into a suite that satisfy a criteria, this technique quantifies the fault revealing ability of each test case and
2. Instead of only injecting syntactic mutants, this technique is based upon the three part software failure model, including empirical evidence of execution coverage, altered data states caused by syntactic mutants, and propagation of simulated infections.

We think that quantifying the fault revealing ability of a single test case is potentially significant, particularly when the quantification is based on a coherent model of software failure. *PIE* is computationally expensive, but does not require an oracle or human intervention to produce its histograms. Furthermore, *PIE* is readily adaptable to parallel processing, and can thus exploit the advantages of MIMD architectures.

PIE is a code-based testing technique designed for the final phase of testing. We can certainly apply *PIE* at any stage in the software development phase, but major code revisions may invalidate past *PIE* results.

One potentially important application of the *PIE* histogram technique is in comparing the effectiveness of competing test selection methods. Although theory can establish a hierarchy of the various coverage criteria [6], this hierarchy does not quantify the increased benefits as one applies more comprehensive coverages. Unless the benefits of increased coverage can be measured, it is difficult to judge whether or not the costs are justified. Using *PIE*, we may be able to compare the benefits of two different test selection schemes for a specific program.

References

- [1] RICHARD A. DEMILLO, RICHARD J. LIPTON, AND FREDERICK G. SAYWARD. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [2] WILLAM E. HOWDEN. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.
- [3] BODGAN KOREL. Dynamic Program Slicing. *Information Processing Letters*, October 1988.
- [4] BODGAN KOREL. PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, SE-14(9), September 1988.
- [5] BRIAN MARICK. Two Experiments in Software Testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1990.

- [6] EDWARD F. MILLER, JR. Program Testing: Art Meets Theory. *IEEE Computer*, July 1977.
- [7] LARRY J. MORELL AND RICHARD G. HAMLET. Error Propagation and Elimination in Computer Programs. Technical Report TR-1065, University of Maryland, Department of Computer Science, July 1981.
- [8] L. J. MORELL. A Model for Code-Based Testing Schemes. *Fifth Annual Pacific Northwest Software Quality Conf.*, pages 309–326, 1987.
- [9] L. J. MORELL. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.
- [10] L. J. MORELL. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16, August 1990.
- [11] LARRY MORELL. Personal discussions.
- [12] LARRY JOE MORELL. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.
- [13] A. J. OFFUTT. *Automatic Test Data Generation*. PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.
- [14] A. J. OFFUTT. The Coupling Effect: Fact or Fiction. *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, December 1989. Key West, FL.
- [15] SANDRA RAPPS AND ELAINE J. WEYUKER. Data Flow Analysis Techniques for Test Data Selection. *Proceedings of the 6th International Conf. on Software Engineering*, pages 272–278, 1982. Tokyo, Japan.
- [16] SANDRA RAPPS AND ELAINE J. WEYUKER. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.
- [17] D. RICHARDSON AND M. THOMAS. The RELAY Model of Error Detection and its Application. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.
- [18] TIMOTHY J. SHIMEALL. CONFLICT Specification. Technical Report NPSCS-91-001, Computer Science Department, Naval Postgraduate School, Monterey, CA, October 1990.
- [19] J. VOAS AND K. MILLER. Improving Software Reliability by Estimating the Fault Hiding Ability of a Program Before it is Written. In *Proc. of the 9th Software Reliability Symposium*, Colorado Springs, CO, May 1991. Denver Section of the IEEE Reliability Society.
- [20] J. VOAS, L. MORELL, AND K. MILLER. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

- [21] J. VOAS AND J. PAYNE. A Parallel Propagation Analysis Algorithm. Technical Report WM-91-2, College of William and Mary in Virginia, Department of Computer Science, March 1991.
- [22] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, To appear in 1992.
- [23] M. R. WOODWARD AND K. HALEWOOD. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.
- [24] STEVEN J. ZEIL. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.