A Dynamic Failure Model For Predicting the Impact that a Program Location has on the Program

Jeffrey Voas
NASA-Langley Research Center
Mail Stop 478, Hampton, VA 23665 USA

Abstract

This paper presents a dynamic technique for predicting the effect that a "location" of a program will have on the program's computational behavior. The technique is based on the *three* necessary and sufficient conditions for software failure to occur: (1) a fault must be executed, (2) the fault must adversely affect the data state, and (3) the adverse effect in a data state must affect program output. In order to predict the effect that a location of a program will have on the program's computational behavior, the following characteristics of each program location are estimated: (1) the probability that a location of the program is executed, (2) the probability that a location of the program noticeably affects the program state created by the location, and (3) the probability that the data states created by a location affect the program's output. With estimates of these characteristics for each location in a program, we can predict those locations where a fault can more easily remain undetected during testing, as well as predict the degree of testing necessary to be convinced that a fault is not remaining undetected in a particular location.

Index Terms: Software testing, data state, sensitivity analysis, mutant, fault, failure probability.

1 Introduction

Software testing reveals failures. When an input distribution for a program is known, random testing has several advantages over other validation techniques; it does not rely on extensive analysis such as *proof of correctness*, it replicates operational behavior, and it has a statistical basis. However, testing has drawbacks: any predictions based on black-box random testing depend on an assumed input distribution. If an input distribution for a program changes, any predictions based on the previous input distribution may change dramatically. When testing reveals a failure, it provides little help in locating the fault. Finally, testing requires an *oracle*; since automated oracles are rarely available, human oracles (who require time and may introduce new faults) are required.

The dynamic technique presented in this paper complements software testing. It does *not* identify faults; correctness is never an issue. Instead, the technique identifies "locations" in a program where faults, *if they were to exist*, are more likely to remain undetected from tests. In this technique, a *location* is defined to be either an input statement, output statement, assignment statement, or the <expression> part of a while or if statement.

This technique requires no oracle nor specification, however it does require that inputs are selected at random consistent with an assumed input distribution. Preferably this input distribution will be the *operational distribution*, however there are certain instances where a uniform distribution may be substituted when the operational distribution is unknown and still produce valid results. The technique uses a program's input distribution, syntactic mutants, and changes injected into dynamically created data states to predict a location's ability to cause program failure *if the location were to contain a fault.* The technique can just as easily be applied to modules as programs; if this is done, the technique would predict the ability of a location in a module to cause module failure if the location were to contain a fault.

The technique makes predictions concerning future program behavior by estimating the effect that (1) an input distribution, (2) syntactic mutants, and (3) changed data values in data states have on current program behavior. More specifically, the technique first observes the behavior of the program when (1) the program is executed with a particular input distribution, (2) a location of the program is injected with syntactic

mutants, and (3) a data state (that is created dynamically by a program location for some input) has one of its data values altered and execution is resumed. After observing the behavior of the program under these scenarios, the technique then predicts future program behavior if faults were to exist. These three scenarios simulate the three necessary and sufficient conditions for software failure to occur: (1) a fault must be executed, (2) a data state error must be created, and (3) the data state error must propagate to the output. Therefore the technique is tightly based on the conditions necessary for software failure.

Three analyses compose the technique: *Execution Analysis (EA)* estimates the probability that a location is executed according to a particular input distribution; *Infection Analysis (IA)* estimates the probability that a syntactic mutant affects a data state; and *Propagation Analysis (PA)* estimates the probability that a data state that has been changed affects the program output after execution is resumed on the changed data state. These analyses together form the technique that this paper presents—the technique is termed *PIE* (for *Propagation*, *Infection* and *Execution* analysis).

The remainder of the paper is organized as follows: Section 1 is responsible for presenting the three analyses: Section 1.1 relates and differentiates *PIE* from other software testing techniques, and Section 1.2 provides the algorithms of *PIE*. Section 2 shows (1) how to apply the estimates that the algorithms produce in order to make predictions about where faults can more easily remain undetected and (2) how to quantify the likelihood that a location will reveal a fault if one were to exist. Section 3 contains empirical results showing that the probability estimates that the algorithms produce do accurately reflect the effect that a location has on the program's computational behavior.

1.1 Survey of Related Techniques

Testing plays a significant role in analyzing software for faults. And as stated in the introduction, *PIE* can be used to complement the efforts of software testing even though *PIE* is fundamentally different than software testing.

PIE is a white-box analysis technique, i.e., its results are a function of the syntax and semantics of the code. Further, *PIE* is a dynamic technique—its results are also a function of an input distribution. Other white-box analysis techniques include coverage-based (or syntactic) testing techniques such as statement testing, branch testing, path testing, and

mutation testing. *Statement testing* attempts to execute every statement at least once; *branch testing* requires that each branch be executed at least once; and *path testing* requires that each path be executed at least once; *EA* differs from statement, branch, and path testing because *EA* does not perform testing—it only estimates the probabilities that a particular location is executed. The results of *EA* can be useful, however, to persons performing statement, branch, or path testing, because *EA* estimates the probability of executing a particular location. For instance, if a person attempting to perform statement testing is having difficulty finding an input to execute a particular statement, then the person could look at the estimate of the probability that the statement is executed (from *EA*), and if it turns out that *EA* has produced a tiny estimate for this probability, then the person can decide whether statement testing is still a feasible goal.

Of all testing techniques, *PIE* is most closely related to mutation testing. It is related in the processes employed, not in the information produced. *Mutation testing* [1] is a testing strategy that evaluates inputs by taking a program P and producing n versions(*mutants*) of P, $[p_1, p_2, ..., p_n]$, that are syntactically different from P. If a set of inputs distinguish the mutants created from P, then it is assumed that if the actual program works with those inputs, the program is good. Mutation testing assumes the "competent programmer hypothesis" that states that a competent programmer produces code that is close to being correct, where "close" means only a few syntax changes are required to correct a program. Mutation testing also assumes that faults that interact can be caught with test data that reveals single faults, i.e., fault coupling is ignored [5]. Mutation testing tests input data; good test data kills all mutants.

IA uses mutation testing ideas in the following way: syntactic changes are made to program statements with the requirement that these syntactic changes must have semantic differences. Whereas mutation testing tests inputs, *IA* tests a location's ability to sustain a change in its semantics yet not change the data state. *PA* generalizes the applicability of mutation testing by allowing a data state to be mutated. *PA* tests to determine the frequency with which a change in a data state causes a change in the program's output. In short, *PA* and *IA*'s goal is significantly different from mutation testing's goals.

Another technique that *PIE* is closely related to is error-based testing. *Error-based testing* attempts to define certain classes of errors and the subdomain of the input space that should reveal any error of that class if that error type exists in the program. Morell

[4] proves properties about error-based strategies concerning certain errors that can and cannot be eliminated using error-based testing. Since error-based testing restricts the class of computable functions, the testing achievable by error-based testing is limited as well. This is because error-based testing defines errors in terms of their syntax, as does mutation testing. *PA* advances the concept of error-based testing by defining classes of errors in terms of their semantic effect in a data state.

In summary, *PIE* is a technique that is fundamentally different than software testing. Testing's goal is to detect faults through the production of failures; *PIE*'s goal is to estimate (1) the probability that a location of a program is executed, (2) the probability that a location of a program noticeably affects the data state created by the location, and (3) the probability that a data state created by a location affects the program's output. With these estimates, *PIE* complements software testing.

1.2 Definitions

The *PIE* technique is built in three levels. We begin with basic terminology in Section 1.2.1; then Section 1.2.2 describes the first level; Section 1.2.3 describes the second level, and Section 1.2.4 describes the third level. The third level contains the algorithms that are used in implementing the technique.

1.2.1 General Definitions

We view a *program* as a function g mapping a domain of possible inputs to a range of possible outputs. Another function, f (with same domain and perhaps different range), represents the desired behavior of g, and can be thought of as a functional specification of g. In practice, it is not necessary to have f explicitly, but it is necessary to be able to say whether a particular output of g is *correct* or *incorrect* for a particular input 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 g, $\tau_{g,D}$, given that g's inputs $x_1, x_2, x_3, \ldots$ are drawn at random according to a fixed but known probability distribution D, is the probability that g results in failure for a randomly selected input, x_i. The probability that a specific input x_i is selected is just $D(x_i)$.

A variable is *live* if its current value may eventually affect the output. The program

counter is considered live. Whether a variable is live or not is determined statically; this can be done using dataflow analysis. A *data state* between two consecutive locations (where consecutive is determined dynamically) is a set of mappings between all statically declared and dynamically allocated variables and their values at that point in execution. Also included in each data state are the input that began the execution and the program counter. The execution of a location is considered to be an atomic operation, hence data states can only be viewed *between* locations. An *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion for that location. A data state error is frequently referred to as an *infection*, and these two terms are used 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.

In formalizing this technique, we will regularly mention "data state mutants." Recall that a data state is just a snapshot during execution of the values of all statically declared and dynamically allocated variables. Thus, a data state is a function of the location where it is taken, as well as the program input and the particular iteration of the location. A *data state mutant* is a change to a data state that would normally occur given an input, location, and iteration of that location.

In this paper, we will consider that data state mutants are created in one of two different ways: (1) either a forced change into a value in a data state (during *PA*), or a change that results from executing a syntactic mutant of the location (during *IA*). Note the difference between a data state error and a data state mutant: a data state error is defined with respect to the correct program, whereas a data state mutant is defined with respect to the current program. The current program that we have in our hands that we are applying this technique to may be correct, but we will not know whether this is true.

1.2.2 Definitions for known faults

Section 1.2.2 describes the first level of the *PIE* technique. The first level formalizes (1) the probability that a location is executed, (2) the probability that an infection occurs, and (3) the probability of an infection propagating to the output. To do so, we define three probabilities: (1) the execution probability, (2) the infection probability, and (3)

the propagation probability. These probabilities are defined for a *known* fault, χ, injected into an otherwise correct program, ρ. Though these definitions are not directly usable in a conventional situation (where the faults, if any exist, are unknown), these definitions provide a foundation for understanding how faults affect the computational behavior of a program.

To define execution, infection, and propagation probabilities, we first introduce notation. Recall that this is a dynamic technique that collects information about what occurs during execution under certain circumstances; some of this information concerns the data states that are created as a program executes. It is therefore necessary to be able to uniquely identify a data state according to the input 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). Unfortunately, all of this identifying information complicates the notion but provides precision.

Let S denote a specification, ρ denote a correct version of S, x denote an input, $domain(S)$ be a set of all possible inputs to ρ for which S is defined, D be the probability distribution of $domain(S)$, l denote a program location in ρ, and let i denote a particular execution (or iteration) of location l caused by input x. Hence if $i > 1$, then l is in a loop or in a procedure that is called more than once. Let $\mathcal{B}_{l,\rho,i,x}$ represent the data state that exists prior to executing location l on the ith execution from input $x \in domain(S)$, and let $\mathcal{A}_{l,\rho,i,x}$ represent the data state produced after executing location l on the ith execution from input x. Note that before ρ begins execution on x, $\mathcal{A}_{l,\rho,i,x}$ and $\mathcal{B}_{l,\rho,i,x}$ equal the empty set for all i and all l. The execution of each location causes the data state that succeeds the location to no longer equal the empty set. Thus if after execution on x, some $\mathcal{A}_{l,\rho,i,x} = \emptyset$, then l was not executed an ith time by 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_x 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. The following sets allow us to do so:

$$\mathcal{B}_{l,\rho,x} = \{\mathcal{B}_{l,\rho,i,x} \mid 1 \leq i \leq n_x\}$$

$$\mathcal{A}_{l,\rho,x} = \{\mathcal{A}_{l,\rho,i,x} \mid 1 \leq i \leq n_x\}$$

We further group these sets into a single set for all $x \in domain(S)$:

$$\beta_{l,\rho,domain(S)} = \{\mathcal{B}_{l,\rho,x} \mid x \in domain(S)\}$$

$$\alpha_{l,\rho,domain(S)} = \{\mathcal{A}_{l,\rho,x} \mid x \in domain(S)\}$$

We let f_l denote the function that *is* computed at 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. It is important to talk about the function that is computed at a particular location in order to see how individual locations map one incoming data state to an outgoing data state.

The *execution probability* of a known fault χ in location l is simply the conditional probability that a randomly selected input x will cause fault χ to be executed:

$$\varepsilon_{l,\rho,D} = \Pr[\mathcal{A}_{l,\rho,x} \neq \emptyset \text{ after } \rho \text{ executes on } x \mid x \text{ selected according to } D] \tag{1}$$

The *infection probability* of a known fault χ in location l is the conditional probability that a data state that location l creates becomes infected when location l is executed for a randomly selected input x:

$$\Pr[\mathit{infected}(\mathcal{B}_{l,\rho,x}, \chi, l) \mid \mathcal{A}_{l,\rho,x} \neq \emptyset \text{ after } \rho \text{ executes on } x] \tag{2}$$

where

$$\mathit{infected}(\mathcal{B}_{l,\rho,x}, \chi, l) = \begin{cases} \mathbf{T} & \text{if } \exists y \in \mathcal{B}_{l,\rho,x}\ f_l(y) \neq f_\chi(y) \\ \mathbf{F} & \text{otherwise} \end{cases}$$

(f_l denotes the function that *should be* computed at location l that instead contains fault χ, and f_χ denotes the function that *is* computed by location l when fault χ is inserted into it.)

The *propagation probability* of a known fault χ in location l is the conditional probability that a program will fail given that the fault at l has infected at least one of l's successor data states:

$$\Pr[\rho \text{ fails on input } x \mid \mathit{infected}(\mathcal{B}_{l,\rho,x}, \chi, l)] \tag{3}$$

1.2.3 Definitions for hypothesized faults and infections

When testing begins, the actual faults in a program are not known. Also, we do not know if our program is correct, which was an assumption in the definitions for the previous

three probabilities. In other words, we do not know if we have a correct program ρ with a single fault χ injected at location l. Therefore the definitions for infection and propagation probabilities are not useful. However, estimation can be performed on the program that we *do* have, denoted by P, by "hypothesizing" that a fault or infection exists in P, and then estimating the effect of the "hypothesized fault" or "hypothesized infection." A hypothesized fault is simply a mutant of a location that is similar to those used in mutation testing [1]. This estimation based on a program P that we do have constitutes the second level of the *PIE* technique.

So in level two, we generalize the definitions of the previous probabilities for hypothesized faults, hypothesized infections, and the current program P. The *hypothesized execution probability* is the conditional probability that location l of program P is executed given an input x selected at random according to D:

$$\varepsilon_{l,P,D} = \Pr[\mathcal{A}_{l,P,x} \neq \emptyset \text{ after } P \text{ executes on } x \mid x \text{ selected according to } D] \tag{4}$$

The *hypothesized infection probability* for a location l and hypothesized fault h is the conditional probability that hypothesized fault h produces a data state mutant in $\mathcal{A}_{h,P,i,x}$ for some input x and iteration i. $\mathcal{A}_{h,P,i,x}$ is the data state that occurs immediately after i^{th} execution of h—this data state is sampled after h is injected into l and execution of P begins on input x. The occurrence of a data state mutant is detected by comparing $\mathcal{A}_{l,P,i,x}$ with $\mathcal{A}_{h,P,i,x}$. The hypothesized infection probability for location l, hypothesized fault h, and input x selected at random according to D is:

$$\lambda_{h,l,P,D} = \Pr[\mathit{infected}'(\mathcal{B}_{l,P,x}, h, l) \mid \mathcal{A}_{l,P,x} \neq \emptyset \text{ after } P \text{ executes on } x] \tag{5}$$

where

$$\mathit{infected}'(\mathcal{B}_{l,P,x}, h, l) = \begin{cases} \mathbf{T} & \text{if } \exists y \in \mathcal{B}_{l,P,x}\ f_l(y) \neq f_h(y) \\ \mathbf{F} & \text{otherwise,} \end{cases}$$

and f_h denotes the function computed by hypothesized fault h at location l. The estimate of the hypothesized infection probability is $\hat{\lambda}_{h,l,P,D}$ and is computed according to the algorithm in Section 1.2.4.

The *hypothesized propagation probability* for a location l in P is the conditional probability that if a data state mutant is injected into the data state immediately following l on l's i^{th} iteration, different program output occurs.

$$\psi_{a,i,l,P,D} = \Pr[\text{differing output from } P \text{ occurs on input } x \mid \mathit{perturbed}(a, \mathcal{A}_{l,P,i,x})]. \tag{6}$$

where

$$perturbed(a, \mathcal{A}_{l,P,i,x}) = \begin{cases} \mathbf{T} & \text{if } a \neq \Omega(a, \mathcal{A}_{l,P,i,x}) \\ \mathbf{F} & \text{otherwise,} \end{cases}$$

and $\Omega(a, \mathcal{A}_{l,P,i,x})$ is a function that extracts the value of variable a in $\mathcal{A}_{l,P,i,x}$. The estimate of the hypothesized propagation probability is $\hat{\psi}_{a,i,l,P,D}$ and is computed according to the algorithm in Section 1.2.4

1.2.4 Algorithms for hypothesized infection probabilities and hypothesized propagation probabilities

We are now at the third level of the *PIE* technique. The third level contains the algorithms necessary for estimating the hypothesized propagation probability, estimating the hypothesized infection probability, and estimating the hypothesized execution probability.

It is necessary in these algorithms to sample internal data states. However the sets $\beta_{l,P,domain(S)}$ and $\alpha_{l,P,domain(S)}$ are not computable if there exists an element of $domain(S)$ on which P does not halt. We can, however, randomly select a finite set of inputs X according to D from $domain(S)$, and if on each $x \in X$ P halts, we can compute $\alpha_{l,P,X}$ and $\beta_{l,P,X}$. Although we cannot determine whether a program will halt for a specific input, we can set a time limit for termination, and if termination has not occurred in that interval, we will not include this input in X. This will be the scheme for creating X. Note also that some element, x_i, may occur more than once in X if $D(x_i)$ is large enough to cause it to be resampled when X is created. So with this we have the sets: $\alpha_{l,P,X}$ and $\beta_{l,P,X}$.

Propagation analysis is a technique that estimates the hypothesized propagation probability defined in equation 6. A *propagation estimate* is an estimate of the hypothesized propagation probability. In PA, a hypothesized data state error is created by a perturbation function. A *perturbation function* is a function that takes in a data value of a variable and changes it according to certain parameters. A perturbation function is a mechanism used by PA for upholding the condition $a \neq \Omega(a, \mathcal{A}_{l,P,i,x})$ in the definition of the function, *perturbed.* A variable whose value is changed by a perturbation function is said to have been *perturbed.*

The algorithm for finding $\hat{\psi}_{a,i,l,P,D}$ (see equation 6) is:

1. Set variable **count** to 0,

2. Randomly select an input x in X, and find the corresponding $\mathcal{A}_{l,P,x}$ in $\alpha_{l,P,X}$. Select data state y to be $\mathcal{A}_{l,P,i,x}$.

3. Perturb the sampled value of a if a is defined, else assign a a random value, and execute the succeeding code on both the perturbed and original data states,

4. For each different outcome in the output between the perturbed data state and the original data state, increment **count**; increment **count** if we believe that an infinite loop has potentially occurred (set a time limit for termination, and if execution is not finished in that time interval, assume an infinite loop occurred),

5. Repeat steps 2-4 n times,

6. Divide **count** by n yielding $\hat{\psi}_{a,i,l,P,D}$.

Note that the perturbation functions defined thus far in this research only perturb data values, however research is ongoing in creating perturbation functions to perturb data structures and the program counter. Since faults can create incorrect data structures and miscalculate the program counter as well as create incorrect data values, this research will eventually suggest perturbation functions to perturb data structures and the program counter. Without these additional perturbation functions, PA is an "incomplete" method.

Infection analysis is a technique that estimates the hypothesized infection probability defined in equation 5. The results of infection analysis are a function of the hypothesized faults used at a location and the distribution of $\mathcal{B}_{l,P,x}$. The estimate of a hypothesized infection probability is termed an *infection estimate.*

The algorithm for finding $\hat{\lambda}_{h,l,P,D}$ (see equation 5) is:

1. Set variable **count** to 0,

2. Create a hypothesized fault for location l denoted h,

3. Randomly select an input x from X, and find the corresponding $\mathcal{B}_{l,P,x}$ in $\beta_{l,P,X}$. Uniformly select a data state, y, from $\mathcal{B}_{l,P,x}$.

4. Present the original location l and the hypothesized fault h with data state y and execute both locations in parallel,

5. Compare the resulting data states and increment **count** when $[f_l](y) \neq [f_h](y)$,

6. Repeat steps 3-4 n times,

7. Divide **count** by n yielding $\hat{\lambda}_{h,l,P,D}$.

The choice of which hypothesized faults to use in this algorithm determines the value gained from performing *IA*. The discussion is limited to hypothesized faults for arithmetic expressions and predicates, because this is where most of the empirical results thus far have come from (see Section 3). For arithmetic expressions, the hypothesized faults considered in this research are limited to single changes to a location: (1) a wrong variable substitution, (2) a variable substituted for a constant, (3) a constant substituted for a variable, (4) expression omission, (5) a variable that should have been replaced by a polynomial of degree k, and (6) a wrong operator. For predicates, the hypothesized faults we consider are limited to (1) substituting a wrong variable, (2) exchanging **and** and **or**, and (3) substituting a wrong equality/inequality operator.

Execution Analysis is a technique that estimates a hypothesized execution probability defined in equation 4. An *execution estimate* is an estimate of the hypothesized execution probability. The execution estimate of location l is denoted $\hat{\varepsilon}_{l,P,D}$ and is computed according to the following algorithm. The input distribution used during *EA* is the same distribution that is used for creating the data state distributions used by *PA* and *IA*.

The algorithm for finding $\hat{\varepsilon}_{l,P,D}$ for the current program P, location l, and probability distribution of the input domain D is:

1. Set array **count** to zeroes, where the size of **count** is the number of locations in the program being analyzed,

2. Instrument the program with "write" statements that signal when a particular location was executed, making sure repeated locations only print once per input,

3. Execute n inputs according to D on the "instrumented" program; this will produce n strings of location identifiers,

4. Increment the corresponding **count**[l] for each printed identifier stating location l was executed; if location l is executed on every input, then **count**[l] $= n$ at the completion of this step.

5. Divide each element of **count**[l] by n yielding $\hat{\varepsilon}_{l,P,D}$.

1.3 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 D:

1. Set 1: The estimate of the probability that program location l is executed, $\hat{\varepsilon}_{l,P,D}$;

2. Set 2: The estimates of the probabilities, one estimate for each hypothesized fault in $(h_1, h_2, ...)$ at program location l, that given the program location is executed, the hypothesized fault will adversely affect the data state: $(\{\hat{\lambda}_{h_1,l,P,D}, \hat{\lambda}_{h_2,l,P,D}, ...\})$; and

3. Set 3: The estimates of the probabilities, one estimate for each variable in $(a_1, a_2, ...)$ at program location l, that given that the variable in the data state following location l is changed, the program output that results from this will too be changed: $(\{\hat{\psi}_{a_1,i,l,P,D}, \hat{\psi}_{a_2,i,l,P,D}, ...\})$.

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 the value of the ns that are chosen in each algorithm. For example, for 95% confidence, the confidence interval is approximately $p \pm 2\sqrt{p(1-p)/n}$, where p is the $\frac{\text{number of occurrences of some event A}}{\text{total number of attempts of event A}}$ (p is the sample mean) [6, 3].

2 Sensitivity Analysis

The remainder of the paper focuses on making predictions. Remember that we want to show how the information of *PIE* can complement software testing. Section 2 shows (1) how to apply the estimates that the algorithms produce in order to make predictions about where faults can more easily remain undetected and (2) how to quantify the number of tests necessary to be convinced that a location is not protecting a fault from detection. Note the shift in emphasis in the paper—from estimation, that *PIE* performs, to prediction, that "sensitivity analysis" performs.

Sensitivity analysis (SA) is a tool for making predictions concerning where faults can more easily remain undetected during testing. Sensitivity analysis uses *PIE*'s estimates to predict the *minimum* effect on the failure probability that a particular location will have, i.e., SA quantifies the minimum effect that a program location has on the program. Note that program computational behavior is not dependent on an oracle; failure probability *is* dependent on an oracle.

With sensitivity analysis, a framework is created that can begin to answer the following questions: (1) Where can we get the maximum benefit from limited testing resources? (2) When should we use another validation technique other than testing? (3) What degree of testing must be performed to persuade ourselves that a location is probably not protecting a fault from detection? (4) When should we rewrite the software in a manner that makes it less likely to protect faults from detection?

Before we begin formalizing sensitivity analysis, consider the following analogy. If software faults were gold, then testing would be gold mining. The results of sensitivity analysis would be a geologist's survey performed before mining begins. It is not the geologist's job to dig for gold, but instead to establish the likelihood that digging at a particular spot would produce gold. A geologist might say, "This valley may or may not have gold, but if there is gold, it will be in the top 50 feet." At another location, the geologist might say, "Unless you find gold in the first 10 feet on this plateau, there is no gold. However, on the next plateau you will have to dig 100 feet before you can be confident that there is no gold."

When software testing begins, such an initial survey has obvious advantages over testing blind. From this analogy, sensitivity analysis is the "geologist." Sensitivity analysis predicts the required testing intensity for a particular confidence, whereas the geologist predicts the digging depth. Sensitivity analysis provides the degree of difficulty that will be incurred during random black-box testing of a particular location to detect a fault. If after random black-box testing to the degree specified by sensitivity analysis only to observe no failures, we then feel confident that the location is *correct.*

Sensitivity of a location l is a prediction of the minimum probability that a fault in l will result in a software failure under a particular program input distribution. If location l is assigned a sensitivity of 1.0 under a particular input distribution D, then it is predicted that each input in D that executes l will result in a software failure if l were to contain

a fault. If l is assigned a sensitivity of 0.0 under D, then it is predicted that no matter what fault is present in l, no input in D that executes l will cause a failure. (Note that there is a continuum of sensitivities in [0,1].) The greater the likelihood that a fault in location l will be revealed during testing, the greater the sensitivity that is assigned to l. A location with a low sensitivity is termed *insensitive*. A location with a high sensitivity is termed *sensitive*.

Testing either reveals or does not reveal faults; *SA* quantifies the significance of testing when testing reveals no faults. If testing's goal is to estimate the probability of failure, sensitivity *is not* an issue. Sensitivity *is* only an issue when testing's goal is to reveal faults. *SA* allows us to gauge how much trust we can place in testing for faults.

As an example, consider a simple program P:

```
{specification:  output 1 if a^2 + b^2 + c^2 < 900000 else output 0}
(1) read(a);
(2) read(b);
(3) read(c);
(4) d := sqr(a) + 1000;
(5) e := sqr(b);
(6) f := sqr(c);
(7) if ((d + e + f) < 900000) then
(8)     writeln("1")
    else
(9)     writeln("0");
```

P is supposed to perform the function described in the brackets but contains a fault in location 4. Assume that testing P under a particular input probability distribution D produces no failures. What does this testing say about the existence of faults in P? While we can make predictions about P's probability of failure under D, we do not have any assurances about an absence of faults in P. This is because: (1) D may not execute portions of P where the faults (if any) reside, (2) incorrect data states may not be produced, or (3) incorrect data states may be cancelled.

For P, Figure 1(i) and Figure 1(ii) contain input probability distributions that are not likely to reveal the fault in location 4; If the range of potential input values for variables **a**, **b**, and **c** are fixed in the interval [545, 550], then the fault is more likely to be caught during testing. *SA* would warn that locations 4, 5, and 6 have the capacity to protect faults from detection if testing is performed according to Figure 1(i) or Figure

Figure 1: Probability distributions that are unlikely to reveal fault.

1(ii) (assuming D = Figure 1(i) or D = Figure 1(ii) when *PIE* is performed).

PIE's probability estimates for P given that D = Figure 1(i) or D = Figure 1(ii) follow:

1. EA reveals that there is a zero probability of a fault existing in locations 4, 5, and 6 and not being executed, and thus produce $\hat{\varepsilon}_{4,P,D} = 1.0$, $\hat{\varepsilon}_{5,P,D} = 1.0$, and $\hat{\varepsilon}_{6,P,D} = 1.0$.

2. IA reveals that locations 4, 5, and 6 produce high infection estimates, suggesting that actual faults in the locations would almost certainly produce infections.

3. Unlike the high probability estimates of IA and EA, PA tells something quite different. As an example, if the value of variable **d** is modestly altered after location 4, it is likely that P will still produce correct output. This is true of variable **e** after location 5, and variable **f** after location 6. PA will take this into consideration and indicate that locations 4, 5, and 6 are locations where infections can more easily remain undetected from testing, i.e., PA will produce low propagation estimates for variables **d**, **e**, and **f** at locations 4, 5, and 6.

Mapping the probability estimates for a location to a single sensitivity for that location is difficult, because determining the relative importance for each different set of probability estimates is difficult. Each estimate has an associated confidence interval, and we take the lower bound on the confidence interval. This assures that if bias occurs when finding a sensitivity, the bias causes underestimation of the sensitivity rather than overestimation.

This is clearly a conservative method. This conservative approach is taken one step further by only considering the minimum of the lower bounds of the infection estimates and the minimum of the lower bounds of the propagation estimates of a location when determining the location's sensitivity.

Let $(\cdot)_{min}$ denote the lower bound for the confidence interval for an estimate, and let $(\cdot)_{max}$ denote the upper bound for the confidence interval for an estimate. The scheme for mapping *PIE*'s probability estimates into a sensitivity prediction of some location l, denoted by θ_l, follows:

$$\theta_l = (\hat{\varepsilon}_{l,P,D})_{min} \cdot \sigma(\min_h[(\hat{\lambda}_{h,l,P,D})_{min}], \min_a[(\hat{\psi}_{a,i,l,P,D})_{min}]) \tag{7}$$

where

$$\sigma(a,b) = \begin{cases} a-(1-b) & \text{if } a-(1-b) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Equation 7 represents a prediction of the minimum probability that "if a fault were to exist in l, the existence of the fault will be revealed through testing."

With sensitivities in hand, we now return to the four proposed questions and explain how *SA* can begin to answer them.

1. *Where to get the most benefit from limited testing resources:*
   Sensitive locations require less testing than insensitive locations. By identifying sensitive locations, sensitivity analysis saves resources that can be applied to more critical insensitive locations.

2. *When to use some other validation technique other than testing:*
   Sensitivity analysis may show extreme insensitivity (frequent protecting of faults from detection), thereby pinpointing locations for which no reasonable amount of testing under an assumed distribution can be performed to gain confidence in such locations' lack of faults. At such locations, alternative techniques should be applied such as testing under a new distribution, proofs of correctness, code review, or exhaustive testing.

3. *The degree to which testing must be performed in order to be convinced that a location is probably not protecting a fault from detection:*
   Sensitivity analysis results may be used to determined how many test cases are

necessary to be convinced a location is correct with an acceptable confidence. θ_l can be used as an estimate of the minimum failure probability for location l in the equation $1 - (1 - \theta_l)^T = c$ [2], where c is the confidence that the actual failure probability of location l is less than θ_l. With this equation, we can obtain the number of tests T needed for a particular c. To obtain confidence c that the true failure probability of a location l is less than θ_l given the sensitivity of the location, we need to conduct T tests, where

$$T = \frac{\ln(1-c)}{\ln(1-\theta_l).} \tag{8}$$

When $\theta_l \approx 0.0$, we effectively have confidence c after T tests that location l does not contain a fault. To obtain confidence c that the true failure probability of a program is less than θ_l given the sensitivities of its locations, we need to conduct T tests, where

$$T = \frac{\ln(1-c)}{\ln(1-\min_l[\theta_l]).} \tag{9}$$

When $\min_l[\theta_l] \approx 0.0$, we effectively have confidence c after T tests that the program does not contain a fault. Note that for equation 8 and equation 9, θ_l can not be zero or one.

4. *Whether or not software should be rewritten:*
   Sensitivity analysis results may be used as a guide to whether critical software has been sufficiently tested. If a piece of critical software is classified as having many insensitive locations, then the software may be rejected since too much testing will be required to achieve a sufficient level of confidence from testing.

3 Results

This section shows that *PIE*'s probability estimates are correlated to the actual effects that program locations have on the program's failure probability *when the locations were forced to contain faults*; this section contains both (1) empirical results and (2) examples. Section 3's results only expose the accuracy of the algorithms; a preliminary experiment showing that equation 7 is conservative is found in [9].

For the empirical results, we took the gold-version, G, of a battle simulation that was approximately 2000 lines in length and is specified in [7]. We

1. Made a copy of G, denoted by G',

2. Randomly selected some location l of G,

3. Injected a single fault F that virtually always infected into G' at location l (typically a fault was just a changed operator in an arithmetic expression),

4. Found $\hat{\tau}_{G',D}$ ($\tau_{G',D}$ represents the failure probability resulting from an injected fault F at location l in G', and $\hat{\tau}_{G',D}$ is an estimate of this probability),

5. Found $\hat{\varepsilon}_{l,G',D}$,

6. Found $\hat{\psi}_{a,i,l,G',D}$, where a is the variable on the left-hand side of the assignment statement at location l, and variable a was forcefully perturbed on *each* iteration i of l. In this experiment, $\hat{\psi}_{a,i,l,G',D}$ is a function of:

   (a) A perturbation function producing a uniformly selected value in the interval $[0.5x, 1.5x]$, where x is the original value variable a had before it was perturbed,

   (b) A uniform program input distribution,

   (c) 100 program inputs, and

   (d) Perturbation functions that were applied on each iteration i of l (if location l was executed more than once for some x).

7. Removed F from G', and

8. Went back to Step 2 and reperformed this analysis on a different l.

In these empirical experiments, we purposely injected faults with infection probabilities of approximately 1.0 so that the likelihood of low infection probabilities affecting the failure probability was negligible. This allowed correlation between $(\hat{\varepsilon}_{l,G',D} \cdot \hat{\psi}_{a,i,l,G',D})$ and $\hat{\tau}_{G',D}$ (see Table 1 and Table 2). The reason for multiplying the execution estimate and the propagation estimate is because propagation estimates are conditioned on executing a specific location; failure probability estimates are not. The correlation found from these 9 experiments is 0.995, meaning that the correlation found between the product of the propagation estimate and execution estimate and failure probability estimate was *high*. This preliminary result supports the claim that PA and EA produce probability

estimates that are accurate enough to predict what effect a particular location has on the computation of the program.

Sixteen similar experiments using the same gold-version resulted in an overall correlation of 0.975 for all 25 experiments. The additional 16 injected faults were all *omission faults*, suggesting that PA can be used under certain circumstances as a debugging technique for omission faults.

For brevity, we will not include empirical results concerning IA and the gold-version, and will instead argue that IA's importance can be easily shown. There are locations that when injected with faults, do not affect the resulting data state; these are locations where faults can more easily remain undetected. Consider a correct location such as **k := i *3**, that is replaced by **k := i * c**. If **i** frequently has the value of 0 or **c** frequently has the value of 3 immediately before the location is executed, infection is not likely to occur. As another example, consider a correct location **x := x mod 1000**. If **x** is typically small ($x < 500$) before this location is executed, then a fault such as **x := x mod 10000**, if injected, would easily go undetected.

4 Concluding Remarks

This paper has presented two techniques that are based on the three necessary and sufficient conditions for software to fail; one technique, *PIE*, performs estimation, and the other, sensitivity analysis, performs prediction.

The first technique dynamically estimates program characteristics that affect a program's computational behavior. This technique does not require a specification nor oracle, however it requires an input distribution. *PIE* may be performed on incorrect programs and still reveal useful information. Since PIE is a code-based technique, it is expected that the program under analysis is "close" to being a correct implementation of the specification, both syntactically close and semantically close.

To date, *PIE* has only been applied to small programs, since no automated PIE system has yet been completed, and all results shown in this paper have been a combination of both manual and dynamic effort. In the future, *PIE* will be performed automatically—work on building a *PIE* system has begun. Once available, we recommend that *PIE* first be applied to the critical portions of a large-scale system. If resources allow for *PIE* to

trial	*location l*	*fault F*
1	XCorner := Batts[Beta][G].X - (((Army[Beta][G].Grow-1)/2)* Army[Beta][G].Squadsep);	XCorner := Batts[Beta][G].X - (((Army[Beta][G].Grow+1)/2)* Army[Beta][G].Squadsep);
2	A4 := (m+1)*((n+1)*Terrain[m,n]-n* Terrain[m,n+1])-m*((n+1)* Terrain[m+1,n]-n*Terrain[m+1, n+1]);	A4 := (m)*((n+1)*Terrain[m,n]-n* Terrain[m,n+1])-m*((n+1)* Terrain[m+1,n]-n*Terrain[m+1, n+1]);
3	Dist := sqrt(abs(sqr(x1-x)+ sqr(y1-y)));	Dist := sqrt(abs(sqr(x1-x)* sqr(y1-y)));
4	C1X := TX-TW/2;	C1X := TW-TX/2;
5	x1 := x+Batts[Beta][G].v* cos(Army[Beta][G].Theta);	x1 := x+Batts[Beta][G].v* sin(Army[Beta][G].Theta);
6	i := trunc(r);	i := trunc(round(r));
7	Xcorner := Batts[Beta][G].X- (((Maxi-1)/2)* Army[Beta][G].Squadsep);	Xcorner := Batts[Beta][G].X- (((Maxi-1)*2)* *Army[Beta][G].Squadsep);
8	TempI := 1+(k+m) mod Batts[Beta][G].NW[E,J];	TempI := (k+m) mod Batts[Beta][G].NW[E,J];
9	Af := Army[Beta][G].FixRate* Batts[Beta][G].Numfixers/cc;	Af := Army[Beta][G].FixRate+ Batts[Beta][G].Numfixers/cc;

Table 1: Injected Faults

trial	*variable a*	$\hat{\varepsilon}_{l,G',D}$	$\hat{\psi}_{a,i,l,G',D}$	$\hat{\tau}_{G',D}$
1	`XCorner`	1.0	1.0	1.0
2	`A4`	0.98	1.0	0.85
3	`Dist`	1.0	0.98	1.0
4	`C1X`	0.98	0.0	0.08
5	`x1`	0.98	1.0	0.98
6	`i`	1.0	1.0	0.98
7	`Xcorner`	0.16	0.8125	0.09
8	`TempI`	0.01	1.0	0.01
9	`Af`	1.0	0.87	0.94

Table 2: Propagation Analysis Results

be performed on non-critical portions, that is beneficial as well.

The reason we mention performing *PIE* on specific program sections is *PIE*'s enormous costs: the sequential PA algorithm in this paper is of quadratic order. Therefore for large-scale systems, probably only sections of the program can have this analysis performed. One way of improving PIE's costs is parallelization; parallelization of the PA algorithm has been shown to produce near linear speed-ups [10].

When to apply *PIE* in the software life-cycle is an important concern—towards the end of the testing and validation phase appears to be the most appropriate time. Once program structure is certain to minimally change, which should be true at this point in the life-cycle, then *PIE* can be applied. If *PIE* were applied at the beginning of the testing phase, and major modifications were made to the program, then *PIE* would almost certainly need to be reperformed.

The second technique predicts whether locations are likely or unlikely to reveal faults. This technique, like *PIE*, is a function of the input distribution that *PIE* uses. It is expected that this input distribution will be the same one that the program will be tested according to. If the input distribution were to change, for example, to an input distribution based on some white-box analysis, then *SA* would predict the likelihood of faults remaining undetected if the program were tested according to this other input distribution. Thus for better predictions from *SA*, *PIE* should use the input distribution that the program is expected to be tested according to, which preferably is the operational distribution.

With *SA*, not only can we say that a program executed k inputs successfully, but if the program has many locations that are sensitive, we keep the confidence gained about the correctness of the program after testing is complete. If a program has many locations that are insensitive, we should realize that although it is possible that the program is correct, it is also possible that faults are remaining undetected, and thus the k successful tests do not offer an equivalent confidence.

Insensitive code is not necessarily bad. After all, correct code can be insensitive. The problem with insensitive code is that it suggests a greater ability to protect faults from detection, and that arguably is not a desirable characteristic. Research has been suggested that certain computable functions may have the misfortune of tending to result in insensitive code when implemented [8, 11]. This says that upper bounds may exist

on the sensitivity that we can achieve for particular functions. We believe that there are design techniques that can be specified to create software that is "generally" more sensitive [11], however these techniques are only preliminary ideas requiring formalization.

5 Acknowledgements

The author expresses gratitude to Larry Morell for the help provided in formalizing these ideas in the years 1988 through 1990. This research was performed before the author accepted his current position as a National Research Council NASA/Langley Resident Research Associate, and during that time was funded as a graduate student under NASA grants NAG-1-824 and NAG-1-884 at the College of William and Mary.

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] Richard G. Hamlet. Probable Correctness Theory. *Information Processing Letters*, pages 17–25, April 1987.

[3] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Book Company, 1982.

[4] Larry Joe Morell. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[5] L. J. Morell. A Model for Code-Based Testing Schemes. *Fifth Annual Pacific Northwest Software Quality Conf.*, pages 309–326, 1987.

[6] S. K. Park. Lecture notes on simulation, version 3.0. Department of Computer Science, College of William and Mary in Virginia, 1990.

[7] Timothy J. Shimeall. `CONFLICT` Specification. Technical Report NPSCS-91-001, Computer Science Department, Naval Postgraduate School, Monterey, CA, October 1990.

[8] J. Voas and K. Miller. Improving Software Reliability by Estimating the Fault Hiding Ability of a Program Before it is Written. In *Proceedings of the 9th Software Reliability Symposium*, Colorado Springs, CO, May 1991. Denver Section of the IEEE Reliability Society.

[9] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[10] 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.

[11] J. Voas. Preliminary Observations On Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, Portland, OR, To appear October 1991. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.