

Applying A Dynamic Testability Technique To Debugging Certain Classes of Software Faults*

Jeffrey M. Voas

Keith W. Miller

Abstract

Testability, the tendency for software to reveal its faults during testing, is an important issue for verification and quality assurance. But testability can also be used to good advantage as a debugging technique. Although this concept is more general, we will illustrate it with a specific example: propagation analysis.

Propagation Analysis (PA) is a technique for predicting the probability that a data state error affects program output. PA is a technique that produces information about a piece of software's testability. PA bases its prediction on empirical measurement of the probability that an "artificial" data state error affects program output. After obtaining propagation analysis information for a program and obtaining a failure probability estimate for the program during execution, we build a model that can be used to identify possible sites of missing-assignment faults of the form $x := f(x)$. Thus we can apply the testability technique PA as a debugging tool.

Keywords: Propagation analysis, software testing, fault, failure, probability of failure, oracle, debugging, fault-based testing.

1 Introduction

Testability, the tendency for software to reveal its faults during testing, is an important issue for verification and quality assurance [15]. But testability can also be used to good advantage as a debugging technique. Although this concept is more general, we will illustrate it with a specific example in this paper: during debugging, we can use

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

information about how faults are likely to propagate through a program to narrow our search for missing assignment faults.

Code-based testing schemes are often criticized for not detecting missing-code faults. Howden [4], for example, showed that detecting missing-path faults is particularly difficult. This paper shows how one code-based testability technique, *propagation analysis* (or *PA*), can be used to suggest whether missing-assignment statements of the form $x := f(x)$ are occurring in a program and suggest places in the code where the missing code is likely to belong. Thus the *PA* testability technique can be used in a debugging model for suggesting where faults may be residing that are resulting in incorrect output. (This is not to be confused with debugging techniques that trace run-time errors or produce compilation error warnings.) We can generalize this debugging model to other classes of faults.

To discover why some code-based testing schemes can detect missing-code faults while others cannot, we need to separate two issues:

1. The criterion whereby test data are selected, and
2. The process whereby test data cause a program to fail.

A *criterion* for test data selection identifies some characteristic of the program to be covered by the test data. Statement coverage and data flow path coverage are two well known examples of test data criteria [16].

The *process* of program failure characterizes how software behaves when it exhibits incorrect input/output behavior. For our analysis, we emphasize three aspects of a software failure. These aspects form the *fault/failure model* that is described in [13]:

1. The faulty code (including where missing-code belongs) must be executed;
2. The data state must be changed in a way that renders some variable incorrect (“infection” of the data state);
3. The incorrect data state must cause an incorrect output (“propagation” to the output).

Although the fault/failure model is simple, its implications are far-reaching, and many code-based testing schemes do not take all three aspects into consideration. In the next section, we describe how several other testing techniques relate to the fault/failure model and to missing-assignment faults. Later, we demonstrate that *PA*, based on the three part fault/failure model, was effective in experiments involving missing-assignment statements.

1.1 Previous Code-Based Testing Techniques

Statement testing, weak mutation testing, data flow testing, and symbolic fault-based testing represent approaches that attempt to make testing more likely to result in failure if faults exist. In this section, we describe how each approach relates to the fault/failure model, and how each interacts with missing-assignment faults. Throughout, the term *test set* refers to tests that fulfill a particular input selection criterion.

Statement testing requires a test set that executes every statement in the program at least once. Statement testing focuses on part one of the fault/failure model—reaching fault locations—but ignores the creation of program state errors and the propagation of program state errors into the output. If the fault is missing-code, then statement coverage ensures the fault location is executed, but it cannot ascertain whether the fault is likely to infect or likely to propagate.

Fault-based testing evaluates inputs based on their ability to distinguish the specific faults. *Mutation testing* [2, 3, 9, 10] is a fault-based testing strategy that does just this—it evaluates program inputs. 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* [2] is to find a set of inputs that distinguishes the mutants from P . Another variant of mutation testing, *weak mutation testing* [3], selects inputs that cause all imagined infections to be created by a possibly infinite set of mutants. Weak mutation testing focuses solely on the creation of alterations in data states. In weak mutation testing, a test set is selected that is believed will cause faults to infect the succeeding data states. Weak mutation testing does not take into account the propagation of these data state errors to output variables; weak mutation testing is only concerned with the creation of data state alterations by syntactic mutants. This leaves the possibility that the data state errors will be cancelled by the ensuing computation. Data state error *cancellation* occurs whenever the existence of a data state error (that is created by a fault) is observed during execution and at a later time during execution, analysis of the data state does not indicate that the original data state was in error.

Data flow testing addresses propagation, since data flow can track the propagation of errors through the program and from one infected variable to another [5, 12]. Since its analysis deals with define-use (“def-use”) pairs in the code, data flow testing has not been routinely applied to missing-code faults. It could, however, be extended to do so if hypothetical def-use pairs were investigated; a hypothetical def-use pair is one that is based upon assumed missing-code—a “what if” analysis [8]. Unfortunately, the number of possible hypotheticals can explode combinatorially in a large program.

Symbolic fault-based testing [7] is a static method that uses symbolic execution to embody the full fault/failure model. A “symbolic fault” is introduced into the code

to represent a hypothetical fault. The resulting code is then symbolically executed, producing a symbolic output description. Symbolic testing enables direct investigation of missing-code faults, but at a sizable cost: the impact of missing-code faults must be determined by the symbolic execution system on a path-by-path basis for each fault envisioned. And for most programs, the number of unique paths is intractable making symbolic fault-based testing limited in this use.

2 The Propagation Analysis Testability Technique

2.1 Preliminary Assumptions

Before we describe *PA*, we assume several properties about any program undergoing *PA* as well as knowledge about the program's 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* [2]. If this property is not met, *PA* analysis will still deliver propagation estimates; however, the estimates will be of less significance. The closeness is required because the confidence in the applicability of the resulting estimates diminishes as we move further away from the assumption.
2. A distribution of inputs, D , is available from which we can sample.
3. The inputs 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.
5. In order to apply the debugging model after *PA* is completed, it will be necessary to know whether software failures are occurring. *PA* does not require an oracle; however to determine whether failure occurs does require an oracle.

2.2 The PA Algorithm

Propagation analysis (*PA*) is a dynamic, code-based analysis method that is based on the full fault/failure model, but which avoids path-by-path analysis. *PA* estimates the impact that a dynamically created artificial data state error will produce on a program's output. When a failure is observed during testing, the estimates gathered using artificial infections can help those debugging the code locate actual faults.

We view a program as an implementation of a function g that maps a domain of possible inputs to a range of possible outputs. Another function f with the same domain and perhaps different range represents the desired behavior 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 a program 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 P , with respect to an input distribution D , τ_{PD} , is the probability that P produces an incorrect output for an input selected at random according to D .

A *location* is loosely defined to be an executable source line. We consider any piece of code that potentially changes a variable or reassigns the program counter (other than a single instruction increment) as a location. For example, `if a < b then` and `a := b` are each a location. A statement like `read(a, b)` embodies 2 locations. A variable is *live* if its current value may potentially affect an assignment or the flow of control in a way that affects the output. The program counter is live.

A *data state* maps variables to values. A data state only occurs between locations because the execution of a location is considered an atomic operation. An *infection* (or *data state error*) is an incorrect variable/value pairing in a data state. If an infection exists, the data state and affected variable at that point are termed *infected*.

If there exists at least one input 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 an input 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.

It is difficult to rigorously identify a specific fault in a program, since infinitely many correct programs exist for any specification. However, for the purpose of this paper we must have some guideline and thus we create the following definition: a program contains a *missing-assignment fault* if the addition of one assignment to the current program decreases the number of faults in the program. We make the assumption that a missing-assignment fault always infects the data state when its absence is executed. By this, we mean executing both locations adjacent to the location where a missing assignment would have to be introduced into the program to decrease its number of faults.

We now introduce notation. Let S denote a specification, P denote an implementation

of S , x denote a program input, Δ denote the set of all possible inputs to P , D denote the probability distribution of Δ , l denote a program location in P , and let i denote a particular execution (or what we term an “iteration”) of location l caused by input x . Let \mathcal{B}_{lPix} represent the data state that exists prior to executing location l on the i^{th} execution from input $x \in \Delta$, and let \mathcal{A}_{lPix} represent the data state produced after executing location l on the i^{th} execution from input 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:

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

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

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

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

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 . An algorithm for estimating $\varepsilon_{l,P,D}$ is described in [15] and we term an estimate of this probability as an *execution estimate*.

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}_{lPix} 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* ψ_{ailPD} 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.

Propagation analysis estimates propagation probabilities. The *propagation estimate*

of propagation probability ψ_{ailPD} is denoted by $\hat{\psi}_{ailPD}$ —it is found by the following algorithm:

1. Set variable **count** to 0.
2. Randomly select an input x according to D , and if P halts on x in a fixed period of time, and find the corresponding \mathcal{A}_{lPx} in $\alpha_{lP\Delta}$. Note that in practice, we expect that P halts on each input x in a fixed (and reasonable) amount of time. If for any input x , P does not halt in this allocated time, we ignore x and will not use any data state \mathcal{A}_{lPix} in this algorithm. ([15] explains how to handle the possibly non-recursive and infinite nature of $\alpha_{lP\Delta}$). Set \mathcal{Z} to \mathcal{A}_{lPix} .
3. 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 the value of variable a are discussed below.
4. For each different result in program output after termination on $\check{\mathcal{Z}}$ and \mathcal{Z} , increment **count**; increment **count** if a time limit for termination related to the altered state has been exceeded. The time limitation should be a function of the time for completion required using the non-altered state \mathcal{Z} . (By assuming that P halts on x in Step 2, we know that the program will halt on \mathcal{Z} .) This precaution is necessary because of the effects that altered variables can produce on boolean conditions that terminate indefinite loops. Of course we cannot be certain that termination using the altered state $\check{\mathcal{Z}}$ will not eventually occur, however we must set some time limit or this algorithm might never terminate.
5. Repeat steps 2-4 n times.
6. Divide **count** by n yielding the sample mean of $\frac{\text{number of times that program output differed}}{n}$, this is our $\hat{\psi}_{ailPD}$.

Note that this algorithm is applied for each live variable a and each location l .

In propagation analysis, a simulated infection is created by a perturbation function. The process of injecting a simulated infection is termed *perturbing*. A *perturbation function* is a mathematical function that takes in a data state as an incoming parameter, changes it according to certain parameters that are either input to the function or hard-wired, and produces as output a different data state. A data state that has had a value changed by a perturbation function is said to have been *perturbed*. We only perturb live variables within a data state because we already know from the definition of live that perturbing a variable in a data state which is not live will result in a 0.0 propagation probability.

Perturbation functions can create a wide variety of simulated infections by using a pseudo-random number generator—we use the Lehmer pseudo-random number generator with a fixed initial seed described in [11]. To perturb, we actually insert the necessary code to cause a state perturbation into the program under analysis. We do so by inserting a source-code module containing the pseudo-random number generator into the code under analysis, and place a call to this module from the location where we want a data state perturbed. We send the module the current data state value and the module returns the perturbed data state value. To date, we have only perturbed numeric data state values; perturbing non-numeric data state values is an area of future research.

The decision concerning when during execution to inject a simulated infection is important to the resulting propagation estimates. That is, during which iteration or iterations of a location do we apply a perturbation function? For example, if a location is in a loop that iterates three times, then we can inject a simulated infection on any of the following combinations of iterations: (1), (2), (3), (1, 2), (1, 3), (2, 3), (1, 2, 3). Note that if we do decide to inject a simulated infection on more than one execution of a location, the simulated infection affects the same variable on each iteration. We currently do not perturb on combinations of variables, due to the explosion in the number of potential combinations. This too is an area for future research.

The choice of how and when to apply a simulated infection depends on the type of data state error we are simulating. Recall that propagation analysis simulates the occurrence of data state errors, and it is important that the simulated infection mimic the real-world (i.e., we must simulate the types of data state errors that actual faults create). For example, since faults in a conditional location can affect which branch is taken after the conditional location, we include the program counter as a live variable; this allows us to perturb the program counter by using an enumerated type (whose members are the different locations that could be executed after the conditional location is executed) and randomly selecting a member of that type as the perturbed program counter value.

As another example, if the type of data state error being simulated can be mapped to a type of fault that has a tendency to produce a data state error each time a fault from this class is executed, then a perturbation function is applied in \mathcal{A}_{IPix} for each i . One instance of this is off-by-one faults, which always infect when executed. In general, mapping simulated infections to potential actual faults will not be possible, since potential faults are very difficult (if not impossible) to determine. So in our research experiments we have perturbed on each iteration (meaning we apply a perturbation function to the current data state value even if that value was a result of a previous perturbation) since our experience using an analysis termed “infection analysis” has shown that faults frequently infect on each iteration. (Infection analysis is described in detail in [15].)

As an example of when a fault would not infect, consider the correct statement $x :=$

$x \text{ div } 30$ and the incorrect statement $x := x \text{ div } 29$. If the incoming value of x for this statement were bounded between 30 and 57, infection would never occur. Thus it is a gamble that we take when we perturb each time; however in the absence of ever being able to know exactly what fault we might be predicting the probability of infection, we assume infection always occurs.

To handle perturbing on each iteration of a location, we define a variant of the previously defined propagation probability; this variant handles the case where a simulated infection is injected into a variable on every iteration of a location. ψ_{alPD} is the probability that P 's output differs given that the value of variable a is perturbed in each data state succeeding location l .

Possible distributions for perturbation functions include all continuous and discrete distributions. To date, we have used a uniform distribution because of our lack of knowledge as to which distribution is best if “a best” exists. Also, the uniform distribution has given encouraging results that are presented in §4.

Currently, PA only perturbs numeric integers and reals. This is not a limitation of the idea of perturbing states, but rather currently a practical limitation, as we do not yet know how to best perturb other types of data values. Also, we have not applied PA to complicated data structures; only simple integers and reals. We are researching ways of perturbing complicated structures as well as pointers.

3 Applying PA In our Debugging Model

We begin by applying the propagation analysis algorithm to the problem of detecting missing-assignment faults through illustration. We will also briefly discuss how to generalize the model discussed here to other fault classes. The reader should note that all that we have talked about so far has dealt with estimating behavioral characteristics of programs. We now wish to use these estimated characteristics to make predictions about where faults might be residing in the software. Thus it is important that the reader note the change in emphasis from estimation to prediction.

Assume that a correct program for a specification contains three assignments $S1$, $S2$, and $S3$ in sequence: $[S1;S2;S3]$. If assignment $S2$ is missing, what we have is $[S1;S3]$ and we do not know that $S2$ is missing. PA effectively creates an ensemble of programs similar to $S1;S3$, for example, $[S1;S2';S3]$, $[S1;S2'';S3]$, $[S1;S2''';S3]$, ..., $[S1;S3;S3']$, ..., and so on. PA then compares the output of each “created” program against the output of $[S1;S3]$ for a set of randomly selected inputs and estimates the frequency of these outputs differing. By then comparing these frequencies with the observed failure probability, we can predict the likelihood that a particular location is

causing the observed failure probability.

If PA correctly estimates the impact that a simulated infection has on the output of the program, and if the estimates from simulated infections can be correlated to impacts on the output caused by actual missing-assignment faults, then PA could be used in an automated process for suggesting where missing-assignment faults might be occurring. Note that PA 's information does not assert that code is actually missing, because PA has no information about correctness. The information from PA can only be used to suggest whether such a fault might be residing and where.

Using PA in such a capacity after a failure or failures have occurred requires the following information:

1. $\hat{\tau}_{PD}$, the failure probability estimate for program P given input distribution D , with units failures per execution. An oracle is needed in order to find this probability estimate.
2. $\hat{\epsilon}_{lPD}$, the execution estimate for each location l . This estimates the first condition of the fault/failure model.
3. $\hat{\psi}_{ailPD}$, the propagation estimate for each a and each l . This estimates the third condition of the fault/failure model.

$\hat{\psi}_{ailPD}$ represents an estimate of the impact that variable a has on the program at location l . If a missing-assignment fault should have assigned a value to variable a but did not, a data state error will almost always result, and unless cancellation of this data state error occurs, an increase occurs to the failure probability (i.e., failure occurs).

The predictive debugging model about to be described assumes infection always occurs. Thus we assume that the probability of the second condition of the fault/failure model occurring is 1.0. Since this assumption is generally true for missing assignment faults, the decision to do this is defensible. This means then that no factor is required in this predictive model for estimating the probability of infection, but we must take into account an estimate of the probability of executing the potential missing-assignment statement.

As with all statistics, there is a confidence interval associated with each probability estimate: the propagation estimate, the failure probability estimate, and the execution estimate. We let $(\cdot)_{mean}$ represent the sample mean for an estimate, $(\cdot)_{min}$ denote the lower bound for the 95% confidence interval (found by using $(\cdot)_{mean} - 2\sqrt{(\cdot)_{mean}(1 - (\cdot)_{mean})/n}$) for an estimate, and $(\cdot)_{max}$ denote the upper bound for the 95% confidence interval (found by using $(\cdot)_{mean} + 2\sqrt{(\cdot)_{mean}(1 - (\cdot)_{mean})/n}$) for an estimate [6]. The method for using PA 's estimates to predict whether or not a missing-assignment fault might be

residing in location l and affecting variable a is provided in equation 1 and equation 2:

$$(\hat{\epsilon}_{lPD})_{min} \cdot (\hat{\psi}_{a|PD})_{min} > (\hat{\tau}_{PD})_{max} \longrightarrow$$

no missing-assignment defining a after location l (1)

suggests that an assignment statement is *not* missing at location l that assigns a value to variable a .

$$(\hat{\epsilon}_{lPD})_{min} \cdot (\hat{\psi}_{a|PD})_{min} \leq (\hat{\tau}_{PD})_{max} \longrightarrow$$

potential missing-assignment defining a after location l (2)

suggests that an assignment *is* missing at location l that assigns a value to variable a . Note that these equations do not say that a missing-assignment fault is or is not occurring; rather they say whether or not it appears likely that such is occurring. Our basis for making such a prediction is from the previously observed program behavior that PA has quantified.

Equation 2 can be a particularly useful when the failure probability estimate is small ($< 10^{-4}$). PA has limited debugging value in the following situations:

1. The program is failing because of distributed faults: their effect on a data state is not currently simulated by PA . We are researching how to simulate distributed faults via perturbation functions.
2. Large failure probability estimates: they force virtually all locations to satisfy equation 2,
3. The program is failing because of many different faults combining into a greater overall failure probability. This debugging model will thus suggest more locations as potentially containing a fault, making the model useless. That is why we assume that the program is close-to-correct.
4. Missing-assignment faults that should define variables that are not even declared in the program.
5. Missing-assignment faults that infrequently infect, e.g., when the value the variable should be assigned is the value that the variable almost always has from a different definition.
6. Boolean variables: A boolean variable has two potential values of true and false. This inflexibility makes it difficult to successfully apply PA . This is because boolean

variables often control exit conditions on indefinite loops. And when we change the value of a boolean variable forcefully, we greatly increase the probability that we will create a non-terminating computation.

7. Reuse of a variable for a different purpose: a missing-assignment fault involving the redefinition of a variable (of the form $x := f(y)$) is unlikely to be handled successfully by *PA*. Redefinition can mean a dramatic change to a variable’s value since its semantic role has changed, and *PA* does not “know” this. *PA* is best at estimating the impact of a missing-assignment fault of the form $x := f(x)$.

Despite these limitations, *PA*’s significant advantage over the alternatives for detecting missing-assignment faults is:

- *PA* can predict tiny missing-assignment fault impacts for specific types of missing-assignment faults. Other debugging models become much less effective when the fault impacts are small, because the existence of such faults is not likely to be revealed during normal debugging, but rather during operational usage; *PA* can be effective for faults with this property, because for extremely tiny failure probabilities, *PA* will lessen the number of locations needing review substantially.

We should also mention that generally $(\hat{\epsilon}_{lPD})_{min}$ and $(\hat{\psi}_{alPD})_{min}$ are large probability estimates. Thus whenever we have a tiny $(\hat{\tau}_{PD})_{max}$ and large values for $(\hat{\epsilon}_{lPD})_{min}$ and $(\hat{\psi}_{alPD})_{min}$, our model eliminates many equations as potentially containing a fault.

Equation 1 and equation 2 can be generalized to other classes of faults that virtually always infect the data state on each execution of the fault. This is because the use of a perturbation function simulates the impact on data states of a wide (and potentially infinite) class of faults. The use of a pseudo-random number generator allows for fault classes to be defined not in terms of syntax, but rather in terms of impacts on data state values.

4 Experimental Results

This section summarizes two experiments in which the estimates produced by *PA* using an incorrect program were correlated to the program’s failure probability. The experiments were performed on a SUN SPARC 1+ workstation with 8 mega-bytes RAM. Due to a lack of an automated system to perform *PA*, we currently can not perform *PA* at every location and every variable. This paper contains results from a less exhaustive *PA* analysis; we used locations that we knew contained faults, since we were forced to perform the analysis manually. As long as we can show that the results of the analysis

that we did perform satisfy equation 2, we know that a complete analysis would still have suggested that the locations where the faults actually resided were locations that this debugging model would have suggested as locations at which to consider debugging. Our results are shown in Table 3, with only the 13th trial failing to satisfy equation 2.

This section shows the correlation coefficient between $((\hat{\epsilon}_{lG'D})_{min} \cdot (\hat{\psi}_{alG'D})_{min})$ and $(\hat{\tau}_{G'D})_{max}$, where an assignment statement is missing that should exist adjacent to location l and should assign a value to variable a . If correlation can be established, we know that equation 1 and equation 2 will provide useful information had a complete *PA* been done for all locations and all live variables. This is the premise for the following experiments.

4.1 Experiment I

The first experiment proceeded in two stages:

1. We inject a fault into a correct program. We assume a certain input distribution and establish an estimate of the failure probability of the (now faulty) program via random testing. For the results shown here, we purposely injected faults with probabilities of creating data state errors (infecting) of approximately 1.0 so that the likelihood of low probabilities of creating data state errors affecting the failure probability was negligible. The faults used were missing assignment statements.
2. We perform propagation analysis and find execution estimates for the faulty program, and use the propagation estimates and execution estimates to predict a probability of failure for the location where we injected the fault.

Our hypothesis is that there will be a significant correlation coefficient between the estimate of the probability of failure measured by random testing and the probability of failure predicted by the estimates of *PA*.

The results reported here are based on the gold-version, G , of a battle simulation that was approximately 2000 lines in length and is specified in [14]. We were told that this program was probably an oracle (from T. Shimeall, the developer of the specification from which it was written) when we were supplied with it, since the supplied program had never been known to fail. The experiment proceeded as follows:

1. Make a copy of G , denoted by G' .
2. Randomly select some location l of G .
3. Inject a fault F into G' at location l .

4. Find $(\hat{\tau}_{G'D})_{max}$, the failure probability estimate for G' , using random testing with distribution D . D was also supplied to us when we received G .
5. Find $(\hat{\epsilon}_{lG'D})_{min}$, the execution estimate for location l in G' .
6. Find $(\hat{\psi}_{alG'D})_{min}$, the propagation estimate for location l in G' , where a is the variable on the left-hand side of the assignment statement at l in G . The $(\hat{\psi}_{alG'D})_{min}$ s are a function of:
 - (a) A perturbation function producing a uniformly selected value in the interval $[0.5x, 1.5x]$, where x is the value variable a had before being perturbed. The Lehmer pseudo-random number generator used is described in [11]. In the situation where $x = 0$, we simply produce the value of either 1 or -1.
 - (b) A uniform program input distribution.
 - (c) 100 program inputs.
7. Go back to step 1.

Using missing-assignment faults with infection probabilities of approximately 1.0 allowed us to calculate the correlation coefficient between $((\hat{\epsilon}_{lG'D})_{min} \cdot (\hat{\psi}_{alG'D})_{min})$ and $(\hat{\tau}_{G'D})_{max}$ (see Table 1 and 2 for more specific details of the experiment). We multiply the execution estimate and the propagation estimate because propagation estimates are conditioned on executing a location.

Using simple linear regression, the *PA* estimate and the actual probability of failure pairs yield a coefficient of determination r^2 of 0.99. Of the 13 trials, only the thirteenth would have failed using the protocol described above for selecting candidate locations.

It should be noted that propagation analysis is a technique that can suffer from qualitative errors because we are making rough approximations via perturbation functions. Our confidence in the value of propagation analysis as an approximation technique are based on experimental results; the approximations have been shown experimentally to often reflect the effects that actual faults cause.

Our preliminary results suggests that propagation estimates and execution estimates are accurate enough to predict the effect that actual faults have on the computation of this program most of the time. These results indicate that *PA* can be effective in predicting whether certain missing-assignment faults are causing a particular probability of failure behavior. Furthermore, these estimates may help locate where the missing-assignment belongs.

<i>trial</i>	<i>location l</i>	<i>fault</i>
1	AF := 1/(Army[Beta][G].Squadrons-ND);	omission fault/missing-assignment
2	m2 := Atan ((m2-m1)/(1+m1*m2));	omission fault/missing-assignment
3	Term := Term*Army[not Beta][E].CommJamEff* (Army[not Beta][E].CommJamRadius - Dist(Batts[not Beta][E].X, Batts[not Beta][E].Y, Batts[Beta][G].X, Batts[Beta][G].Y)) / Army[not Beta][E].CommJamRadius;	omission fault/missing-assignment
4	C := C - icy;	omission fault/missing-assignment
5	Tbatts[Beta]G.BK := Tbatts[Beta]G.BK +1;	omission fault/missing-assignment
6	ND := ND + 1;	omission fault/missing-assignment
7	Temp := Temp + Term;	omission fault/missing-assignment
8	XCur := XCur + Army[Beta][G].Squadsep;	omission fault/missing-assignment
9	Temp := Army[Beta][G].mweffect* Abs(Temp-wmaxseverity*numwevents)/ (wmaxseverity*numwevents);	omission fault/missing-assignment
10	YCur := YCur -Army[Beta][G].Rowsep;	omission fault/missing-assignment
11	Result := Result + Term;	omission fault/missing-assignment
12	Utot[J] := Utot[J] + TBatts[Beta][G]. nw[e,j];	omission fault/missing-assignment
13	Result := Abs(Result-params.numwevents* params.wmaxseverity)/ (params.wmaxseverity* params.numwevents))* Army[Beta][G].vweffect;	omission fault/missing-assignment

Table 1: Injected Faults From Experiment I.

<i>trial</i>	<i>a</i> variable	$(\hat{\epsilon}_{IG'D})_{min}$ execution estimate	$(\hat{\psi}_{aIG'D})_{min}$ propagation estimate	$(\hat{\tau}_{G'D})_{max}$ failure probability estimate	$(\hat{\epsilon}_{IG'D})_{min} \cdot (\hat{\psi}_{aIG'D})_{min}$
1	AF	1.0	1.0	1.0	1.0
2	m2	0.96	0.39	0.68	0.37
3	Term	0.96	1.0	1.0	0.96
4	C	0.94	0.068	0.1	0.06
5	Tbatts[Beta]G.BK	1.0	0.97	1.0	0.97
6	ND	0.49	0.02	0.13	0.01
7	Temp	1.0	0.91	1.0	0.91
8	XCur	1.0	1.0	1.0	1.0
9	Temp	1.0	1.0	1.0	1.0
10	YCur	1.0	1.0	1.0	1.0
11	Result	0.19	0.0	0.0	0.0
12	Utot[J]	0.0	0.0	0.0	0.0
13	Result	0.91	0.32	0.21	0.29

Table 2: Probability Estimates from Experiment I.

<i>trial</i>	$(\hat{\epsilon}_{lPD})_{min} \cdot (\hat{\psi}_{alPD})_{min} \leq (\hat{\tau}_{PD})_{max}$
1	yes
2	yes
3	yes
4	yes
5	yes
6	yes
7	yes
8	yes
9	yes
10	yes
11	yes
12	yes
13	no

Table 3: Accuracy of Equation 2

<i>trial</i>	<i>assignment</i>	<i>fault</i>	$(\hat{\epsilon}_{lG'D})_{min}$ (execution estimate)	$(\hat{\psi}_{alG'D})_{min}$ (propagation estimate)	$(\hat{\tau}_{G'D})_{max}$ (failure prob. estimate)
1	$d := \text{abs}(x-p) + \text{abs}(y-q);$	$d := \text{abs}(y-p) + \text{abs}(x-q);$	1.0	0.0	0.0
2	$y := c.b + s*c.dy;$	$y := c.b + s*c.dy + 1;$	1.0	0.99	1.0
3	$d := \text{abs}(x-p) + \text{abs}(y-q);$	$d := \text{abs}(x-p) - \text{abs}(y-q);$	1.0	0.57	0.37
4	$d := \text{abs}(x-p) + \text{abs}(y-q);$	$d := \text{abs}(x-p) * \text{abs}(y-q);$	1.0	0.41	0.46
5	$p := 0;$	$p := 3;$	1.0	0.68	0.75

Table 4: Non-Missing-Assignment Fault Results from Experiment II.

4.2 Experiment II

In the second set of experiments, we use the gold-version of a proportional navigation problem of the specification described in [1]. Our experiment was performed in a similar manner as was Experiment I. Notable differences include the fact that the program was executed with 1000 inputs, instead of 100 as were used in Experiment II, and the fact that with this set of experiments, various perturbation functions were applied, since a uniformly selected value in the interval $[0.5x, 1.5x]$ occasionally caused infinite loops. The program used can most easily be described as numerical, and was approximately 250 lines of code.

Tables 4 and 5 contain the results from this second set of trials. Of the 8 trials, three were missing-assignment faults (See Table 5), and 5 were other semantic modifications of statements that virtually always caused infection (See Table 4). When these 8 trials are added to the 13 shown in Table 2, the simple linear regression coefficient of determination r^2 was 0.97, again suggesting a high correlation.

In Experiment II, there was, however, one notable exception. The third trial resulted in the product of the propagation estimate and execution estimate being substantially higher than the observed failure probability. This would have forced the location actually containing the fault to have been overlooked as a potential location for where the fault

<i>trial</i>	<i>assignment removed</i>	$(\hat{\epsilon}_{iG'D})_{min}$ (execution estimate)	$(\hat{\psi}_{alG'D})_{min}$ (propagation estimate)	$(\hat{\tau}_{G'D})_{max}$ (failure prob. estimate)
6	<code>s := s + c.delta;</code>	1.0	1.0	1.0
7	<code>x := c.a + s*c.dx;</code>	0.99	0.99	0.99
8	<code>y := c.b + s * c.dy;</code>	0.99	0.99	0.99

Table 5: Missing-Assignment Fault Results from Experiment II.

was if the debugging heuristic of §3 had been applied. This is the same situation that occurred on the thirteenth trial in Experiment I.

The reason for occasional failures of the protocol are statistical in nature: *PA* attempts to make a prediction of the impact on the data state based on a potentially infinite class of faults that might reside in a location. Certainly, a specific fault can have an impact far different than the prediction based on such a large class. Despite the possibility of such failures, the experiments thus far suggest that in a majority of actually fault locations, *PA* does has sufficient accuracy to be useful.

5 Conclusions

Propagation Analysis (*PA*) estimates where particular simulated infections can easily hide. To show the utility of applying testability analysis results to software debugging, this paper has used the propagation analysis technique. Our experimental results limited the simulated infections that *PA* injected to those that are similar to the data state errors created by missing-assignment faults of the form $x := f(x)$ where x is a real or integer variable, and to other fault classes that affect the right hand side of an assignment statement and that almost always infect. Experimental results indicate that *PA*'s testability estimates are correlated with the probability of failure resulting from such classes of faults. Thus software testability information can be of limited help to debuggers when a program is experiencing infrequent failures. Additional research on how to simulate data state errors for other classes of faults is ongoing.

PA is a technique that is used to assess software testability. This paper has conjectured that software testability information, whether gathered via *PA* or some other technique, offers valuable insights during debugging. *PA* is computationally expensive, and it may be impractical to use *PA* to locate a missing fault in a very large program. However, the results from our experiments support our contention that testability information is linked to code characteristics that are important during debugging. We expect that an improved understanding of propagation, new software tools, or new ways to measure testability will eventually make testability information a common source of useful insights during debugging.

6 Acknowledgements

We would like to thank George Tan of NASA-Langley for the use of a SUN 4/70, Timothy Shimeall of the Naval Post-Graduate School for providing access to the battle simulations, Larry Morell of Hampton University for insights on previous testing strategies, and Steve Park of the College of William and Mary for reviewing an earlier draft. The authors also thank the anonymous referees for their insightful and valuable comments.

References

- [1] College of William and Mary. *A Proportional Navigation Problem*, Fall Semester, 1990 edition. Computer Science Course CS242 Project.
- [2] 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.
- [3] WILLIAM E. HOWDEN. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.
- [4] WILLIAM E. HOWDEN. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2:208–215, September 1976.
- [5] LANUSZ W. LASKI AND BODGAN KOREL. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.
- [6] AVERILL M. LAW AND W. DAVID KELTON. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.
- [7] L. J. MORELL. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16, August 1990.
- [8] B. W. MURRILL AND L. J. MORELL. Error Flow Testing. Technical Report WM-90-1, College of William and Mary, Department of Computer Science, 1990.
- [9] A. J. OFFUTT. *Automatic Test Data Generation*. PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.
- [10] 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.
- [11] STEPHEN K. PARK AND KEITH W. MILLER. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

- [12] SANDRA RAPPS AND ELAINE J. WEYUKER. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, pages 367–375, 1985.
- [13] 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.
- [14] TIMOTHY J. SHIMEALL. CONFLICT Specification. Technical Report NPSCS-91-001, Computer Science Department, Naval Postgraduate School, Monterey, CA, October 1990.
- [15] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, To appear in 1992.
- [16] S. N. WEISS AND E. J. WEYUKER. An extended domain-based model of software reliability. *IEEE Trans. on Software Engineering*, 14(10):1512–1524, October 1988.