A Dynamic Testing Complexity Metric*

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

Abstract: *This paper introduces a dynamic metric that is based on the estimated ability of a program to withstand the effects of injected "semantic mutants" during execution by computing the same function as if the semantic mutants had not been injected. Semantic mutants include: (1) syntactic mutants injected into an executing program and (2) randomly selected values injected into an executing program's internal states. The metric is a function of a program, the method used for injecting these two types of mutants, and the program's input distribution; this metric is found through dynamic executions of the program. A program's ability to withstand the effects of injected semantic mutants by computing the same function when executed is then used as a tool for predicting the difficulty that will be incurred during random testing to reveal the existence of faults, i.e., the metric suggests the likelihood that a program will expose the existence of faults during random testing assuming faults were to exist. If the metric is applied to a module rather than to a program, the metric can be used to guide the allocation of testing resources among a program's modules. In this manner the metric acts as a white-box testing tool for determining where to concentrate testing resources.*

Index Terms: Revealing ability, random testing, input distribution, program, fault, failure.

---

*Research supported by a National Research Council NASA-Langley Resident Research Associateship. An earlier version of this paper was presented at the ***1991 Annual Oregon Workshop on Software Metrics.***

†Jeffrey Voas is working as a National Research Council Resident Research Associate at the National Aeronautics and Space Administration's Langley Research Center. His research interests include software testability, data state error propagation, debugging, and design techniques for improving software testability. Voas received a BS in computer engineering from Tulane University and a MS and PhD in computer science from the College of William and Mary.

1 Introduction

Software metrics are simply software characteristics that can be quantified. Over 90 different metrics are available in software engineering [?], each existing to reveal a different type of information about the state of the software. For instance, the lines of code metric can suggest to a project manager the syntactic productivity of his programmers over some time interval. McCabe's cyclomatic complexity metric [?] can suggest the syntactic complexity of the code to a tester performing coverage-based testing [?]. Both of these metrics are quantified statically as are most metrics that measure syntactic software complexity [?].

This paper describes a dynamic metric that measures semantic software complexity; in order for this metric to be quantified, the program must be executed. The dynamic software metric that is the focus of this paper is termed "revealing ability." *Revealing ability* predicts a program's ability to allow faults to be undetected during dynamic random software testing.

Software testing can generally be divided into dynamic software testing and symbolic software testing. *Symbolic* software testing uses symbolic representations for input values; it selects a path and creates expressions for the computation of the selected path in terms of the symbolic values [?]. The symbolic expressions can then be compared against equations representing the correct computational expressions for that path. *Dynamic* software testing executes the program on actual inputs that are chosen in one of two ways: (1) either the code is considered as the inputs are chosen (these are termed *white-box* input selection techniques), or (2) the code is not considered as the inputs are chosen (these are termed *black-box* input selection techniques). After the inputs are selected, the program is then dynamically executed on the inputs and the resulting outputs are compared against the *correct* outputs that are assumed to be available. For this metric, regardless of whether the inputs are chosen in a white-box or black-box manner, it is assumed that they will be chosen at random. Random input selection techniques can be implemented using a pseudo-random number generator such as the Lehmer random number generator with a fixed initial seed described in [?].

Since the revealing ability metric is a dynamic metric, it requires inputs. Revealing ability assumes that the inputs will be chosen randomly from the *operational* distribution, which is assumed will be the input distribution used during random testing. However it cannot be assumed that the operational distribution will always be known. When unknown, the input distribution used will be a uniform distribution over all elements of the input domain. In general, substituting input distributions will bias this metric, and research is continuing into how great such a bias will be.

The revealing ability metric is a tool for deciding whether faults can remain undetected

during dynamic random software testing. (From this point on, this is termed random testing for short). The information revealing ability contains is particularly important to *critical* software, since the existence of undetected faults poses the threat of a loss-of-life [**?**]. Since it is likely that critical software will already have such a high reliability that it will not fail at the time when this metric would normally be quantified (towards the end of testing and validation), the revealing ability metric provides additional insight into determining how significant the absence of observed failures is. If the revealing ability metric suggests that faults can easily remain undetected, observing no failure is less significant; if the revealing ability metric suggests that faults can easily be detected, observing no failure is more significant.

Two different programs can compute the same function with the same input distribution and produce different revealing abilities. A program has *high revealing ability* if it is believed to readily reveals faults during random testing; a program with *low revealing ability* is believed to be unlikely to reveal faults during random testing. High revealing ability suggests that a reasonable number of random inputs will expose faults assuming faults were to exist. Low revealing ability is a dangerous circumstance because considerable amounts of random testing may not produce failures even though faults exist.

Not only can the revealing ability metric be quantified for a program, but the revealing ability of a module or an individual program location can also be quantified. A *program location* is what Korel [**?**] terms a single instruction: an assignment, input statement, output statement, and the <expression> part of an **if** or **while** statement. This paper concentrates on finding the revealing abilities of program locations. The revealing ability of a module or program is derived from the revealing abilities of the individual program locations that compose it.

The remainder of this paper is organized as follows: Section 2 describes the role of mutation in the revealing ability metric. Section 3 describes a technique termed "propagation, infection, and execution analysis"—this is a technique that produces information that is needed in order to quantify revealing ability. Section 4 contains the revealing ability formulae.

2 The Role of Mutation in Revealing Ability

The revealing ability of a program location is a function of three characteristics that are dynamically estimated for the program location. These characteristics are: (1) how often inputs execute the program location, (2) how often injected syntactic changes to the program location cause discernible changes in the resulting internal program states, and (3) how often an injected change to a data value in a program state of an executing program is discernible

in the program output. Thus static and dynamic mutation of a program play a large role in the dynamic revealing ability metric.

For dynamic mutation to occur, it is necessary to sample the program states that occur between consecutively executed program locations. Whether program locations are considered as occurring consecutively is calculated dynamically—thus for different inputs, the location that is immediately executed after some location l may vary according to the input being used. A *program state* is set of mappings of all statically declared and dynamically allocated variables to their current values during an execution using a particular program input—program states are sampled only between consecutively executed program locations. In addition to these variables, the program counter and its current value are also included in a data state set. For example, if a particular input causes five program locations to be executed and each program location is executed once, then there are four program states that could be sampled during that execution. If a particular input causes 10 program locations to be executed and each location is executed 5 times, then there are 49 program states that could be sampled. In general, if the total number of location executions is n, then there are $n - 1$ program states.

Mutating source program constructs statically is common in several software engineering techniques, e.g., fault-seeding and mutation testing. In *fault-seeding*, the goal is to seed mutants into a program and see how many mutants can be caught by a fixed set of inputs. If for example, half of the mutants are caught, and that set of inputs has already revealed the existence of 10 faults in the program, then it is predicted that 10 faults still remain in the program. In *mutation testing* [**?**, **?**], mutants are injected into multiple copies of a program to find a set of inputs that can distinguish all of the mutated versions from the original program. The resulting set of inputs is termed mutation adequate, and these inputs are then used during software testing.

Like fault-seeding and mutation testing, this technique also mutates source program constructs creating what are termed *syntactic mutants*. However the information that is collected from the syntactic mutants is different than the information collected by fault-seeding and mutation testing. Another distinction is that this technique mutates dynamically created program states—these are termed *program state mutants*. Program state mutants are program states that have been altered according to specific rules—these rules are discussed in Section 3.1.

3 The Propagation, Infection, and Execution Analysis Technique

The technique used for quantifying the characteristics needed for the revealing ability of a program location is termed propagation, infection, and execution analysis [?, ?]. Propagation, infection, and execution analysis is a dynamic white-box analysis technique. With these characteristics for a particular program location, a prediction for whether random testing of the program location will reveal faults (assuming faults were to exist) is provided. And with these characteristics for every program location, a prediction of whether random testing of the entire program will reveal faults (assuming faults were to exist) can also be derived. Thus a way of determining the semantic testability of the program is available.

Propagation, infection, and execution analysis is a technique that is based on the three necessary and sufficient conditions for software to fail. These conditions are:

1. A fault must be executed,
2. The fault must affect the state of the program in a manner different than what the state of the program would have been had the fault not existed, and
3. The erroneous program state must propagate to an output state.

These three conditions that are necessary for a fault to cause a software failure are simulated by the processes of propagation, infection, and execution analysis. Propagation, infection, and execution analysis estimates: (1) the probability that a program location is executed according to some program input distribution, (2) the probability that an injected syntactic mutant will produce a discernible difference in the resulting program state, and (3) the probability that injected program state mutants will produce a discernible difference in the program's output.

In order to estimate these three probabilities, propagation, infection, and execution analysis is composed of three subtechniques—each technique is responsible for estimating one probability. These three techniques involve significant computational resources. Unlike software testing, however, these techniques do not require an oracle to determine a correct output. Instead, propagation analysis and infection analysis detects *changes* between the original program's behavior and the behavior of the program receiving injected semantic mutants. This is done without determining correctness. This allows propagation, infection, and execution analysis to be automated, however there are feasibility problems that arise in building an automated system. These difficulties are theoretically unsolvable, however

inelegant partial solutions can be applied to lessen the theoretical difficulties, which still allow for an automated system [?, ?].

In order for the propagation, infection, and execution analysis technique to have practical application, the current program is assumed to be close to its correct syntactic and semantic form. This is similar to the *competent programmer hypothesis* [?]. Without this requirement, each major syntactic program modification would require propagation, infection, and execution analysis to be reperformed. This is not a limitation of the technique, but a limitation that must be imposed to make the technique practical, since propagation, infection, and execution analysis is expensive to perform.

3.1 Propagation Analysis

Propagation analysis estimates the probability that an altered program state will *propagate* it's effect to the program output. To make this estimate, a program state is *perturbed* (or altered) by changing the value of one live variable in the program state. A variable is considered *live* if the variable has *any* potential of affecting an output computation. The set of variables that are live is determined statically using data-flow analysis [?]. After injecting a program state mutant, program execution resumes until termination (assuming no infinite loop occurs); the following algorithm tells how to handle the suspected occurrence of an infinite loop caused by an injected program state mutant.

This process of first perturbing a live variable's value and then resuming execution is repeated many times. This repeated process yields the number of instances k out of n attempts where perturbing a live variable a at some program location l made a discernible effect in the program's output. The value $\frac{k}{n}$ is a rough estimate of the probability that an altered value to variable a (at program location l) affects the program output.

The probability that live variable a, whose value is perturbed at program location l, affects program output is denoted by $\psi_{l,a}$. The estimate of this probability is termed a *propagation estimate*; $\hat{\psi}_{l,a}$ denotes this probability estimate and is computed according to the following algorithm:

1. Set variable **count_prop** to 0.

2. Randomly select a program state from the space of program states that occur immediately after program location l—the space of program states used in this algorithm must be a function of the same input distribution to the program that is used during execution analysis (See Section 3.3).

3. Perturb the value of live variable a in this program state, and execute the code that succeeds program location l on both the perturbed and original program states. There are issues that must be addressed concerning how to perturb a value in a data state, but they are out of the scope of this paper. These issues are addressed in [?, ?].

4. If a different result occurs in the program output between the perturbed program state and the original program state, increment **count_prop**; also, set a time limit for termination of the program executing on the perturbed data state, and if execution is not finished in that time interval, increment **count_prop**.

5. Repeat algorithm steps 2-4 n times.

6. Divide **count_prop** by n yielding $\hat{\psi}_{l,a}$.

A propagation estimate is a function of the program location, the live variable, the space of program states occurring after the program location, the method used to perturb a value, and the code that is potentially executed after the program location. Propagation analysis finds a propagation estimate for each member of the set of live variables at each program location. This produces a large set of propagation estimates—one propagation estimate per live variable per program location. For example, if propagation analysis is performed on a program with 10 program locations and are 5 live variables at each program location, 50 propagation estimates would result.

A propagation estimate can be thought of as an estimate of the effect that a live variable at a program location has on the program's output. For instance, if some live variable's value can be changed at some program location l without producing any evidence of this change in the program's output, i.e., a tiny propagation estimate is produced, then confidence can be gained that this variable at this program location has little effect on the program's computation. It may be that this variable has a greater effect on the program's behavior at some other program location.

Program state mutants are created using perturbation functions. A *perturbation function* is a tool that takes in a value of a live variable and changes it according to certain parameters. Perturbation functions can simulate a wide variety of program state mutants by using functions that are based on random distributions. For example, **uniform(0.5*cv, 1.5*cv)** is a perturbation function that produces a uniformly distributed random value between half and one-and-a-half times the current value of the live variable, **cv**.

The perturbation functions used 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 as well as incorrect

data values, future research will suggest perturbation functions that perturb data structures and the program counter.

3.2 Infection Analysis

Infection analysis estimates the probability that a syntactic mutant injected into a program location will will cause a discernible difference in the program state that results when the mutant is executed. In other words, will a particular syntactic mutant of a program location produce a value in the resulting program state that is visibly different than the value that is produced by the original program location?

Infection analysis is a technique that is similar to *weak mutation testing* [**?**]; what is different in these techniques is the information collected. In infection analysis, a set of syntactic mutants is created for each program location. A restriction is placed on each syntactic mutant—the syntactic mutant must also be semantically different for at least one input to the program. This means that there must be at least one program input such that when the syntactic mutant is executed, the resulting program state *is* different.

Without this requirement, a syntactic mutant could be used during infection analysis that is not discernible when the resulting program states are compared. In this situation, such a syntactic mutant is functionally equivalent to the original program location for all inputs given a particular input domain. For the revealing ability metric, such a situation is unacceptable, because it suggests that the program location has a greater ability to protect faults from detection, which cannot be justified from such a mutant. For example, $\mathbf{x := x + 1}$ would not qualify as a syntactic mutant of $\mathbf{x := 1 + x}$.

Infection analysis creates a set of syntactic mutants for each program location. After creating a program location's mutant set, each syntactic mutant and original location is executed with a data state that is selected at random from the space of data states that occurs before the location. This process is repeated; thus many data states are selected. This repeated process yields the number of instances k out of n attempts where injected syntactic mutant p discernibly affects the program state in a manner that is different than the original location. The value $\frac{k}{n}$ is a rough estimate of the probability that injected syntactic mutant p into program location l discernibly affects the program state. The goal is to determine the effect, $\frac{k}{n}$, for each of these syntactic mutants.

The probability that syntactic mutant p affects the program state differently than the original location does is denoted by $\lambda_{l,p}$. An estimate of this probability is termed an infection estimate and is denoted by $\hat{\lambda}_{l,p}$; $\hat{\lambda}_{l,p}$ is computed according to the following algorithm:

1. Set variable **count_inf** to 0.

2. Create a syntactic mutant for program location l denoted as p.

3. Present the original program location l and the syntactic mutant p with a randomly selected program state from the space of program states that occur immediately prior to program location l, and execute both program locations in parallel (the space of program states used here is expected to be a function of the same program input distribution that is used by execution analysis in Section 3.3).

4. Compare the resulting program states and increment **count_inf** when the result computed by p does not equal the result computed by l for this particular program state.

5. Repeat algorithm steps 3 and 4 n times.

6. Divide **count_inf** by n yielding $\hat{\lambda}_{l,p}$.

An infection estimate is a function of the program location, the syntactic mutant used, and the space of program states that occur immediately before the program location.

The set of syntactic mutants created at a program location should be representative of the class of faults that is expected could occur at the program location. A fault is termed "common" if it is one that might occur there. Unfortunately this is a subjective determination and research is continuing into creating syntactic mutants. A set of syntactic mutants used at a program location is considered non-representative if all its members have significantly higher or lower infection estimates than a common fault at that program location.

The choice of which mutants to use determines the benefit gained from infection analysis. And although there are certain instances of syntactic mutants that are fairly easy to find that would probably satisfy most user's definition of representative, in general, the cost of determining whether a particular mutant is representative will be prohibitive. Therefore in practice the utopian idea of generating syntactic mutants that behave as faults that might occur is abandoned, and will instead generate mutants according to specific rules.

The rules use in this research for generating syntactic mutants are limited to syntactic mutants for arithmetic expressions and predicates. For arithmetic expressions, the syntactic mutants considered in this research are limited to the following changes to a program 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 syntactic mutants considered at a predicate program location are limited to: (1) substituting a wrong variable, (2) exchanging **and** and **or**, and (3) substituting a wrong equality/inequality operator.

3.3 Execution Analysis

Execution analysis is the most straightforward and least computationally expensive of the three techniques. *Execution analysis* requires a program input distribution. Execution analysis executes the code with randomly selected inputs consistent with the input distribution and records which program locations are executed by each input. It is preferred that the input distribution used during execution analysis will be the *operational distribution*, since this is the distribution that is expected to be used during program testing. It is required that the input distribution used during execution analysis will be the same distribution that is used to create the internal program state spaces used by infection analysis and propagation analysis.

The probability that a particular program location l is executed by a randomly chosen input from this input distribution is denoted by ε_l. Execution analysis produces estimates of ε_l. This probability estimate is denoted by $\hat{\varepsilon}_l$ and is termed an *execution estimate*; the method for computing $\hat{\varepsilon}_l$ is given in the following algorithm:

1. Set array **count_exec** to zeroes, where the size of **count_exec** is the number of program locations.

2. Instrument the program with **write** statements at each program location that print the program location number when the program location is executed. Make sure that if a program location is repeated more than once for some input, the **write** statement for that program location is only executed once for that input.

3. Execute n randomly selected program inputs on the instrumented program, producing n strings of program location numbers. It is assumed that the program halts for each of these n inputs, however if it is suspected that this will not be true, even with the close to being semantically and syntactically correct assumption, then a process must be inserted to ignore inputs on which execution analysis *appears* to not be terminating and hence select new inputs. This process will be performed until the program is executed to termination on n inputs.

4. For each program location number l in each string, increment the corresponding **count_exec**[l]. If it happens that some program location k is executed on each of the n inputs, then when Step 4 is completed, **count_exec**[k] would equal n.

5. Divide each **count_exec**[l] by n yielding an execution estimate $\hat{\varepsilon}_l$.

An execution estimate is a function of the program, the program location, and a particular input distribution. If the input distribution is changed, then the estimates of execution

analysis, as well as the probability estimates from infection analysis and propagation analysis, will in all likelihood change.

3.4 Understanding the Resulting Estimates

After propagation, infection, and execution analysis has been performed for the entire program, there are three sets of probability estimates for each program location l:

1. Set 1: The estimate of the probability that program location l is executed ($\hat{\varepsilon}_l$);
2. Set 2: The estimates of the probabilities, one estimate for each syntactic mutant in $\{p_1, p_2, ...\}$ at program location l, that given the program location is executed, the syntactic mutant will adversely affect the program state; these are denoted as $(\{\hat{\lambda}_{l,p_1}, \hat{\lambda}_{l,p_2}, ...\})$; and
3. Set 3: The estimates of the probabilities, one estimate for each live variable in $\{a_1, a_2, ...\}$ at program location l, that given that the live variable in the program state following the program location is perturbed, the output will be changed; these are denoted as $(\{\hat{\psi}_{l,a_1}, \hat{\psi}_{l,a_2}, ...\})$.

These parameters are necessary for the revealing ability formulae. 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 propagation, infection, and execution analysis 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 $\frac{\text{number of occurrences of event A}}{\text{number of attempts of event A}}$ (this is just the sample mean) [?, ?]. Although there is an associated confidence interval for each probability estimate, the confidence interval is discarded and use only the mean of the confidence interval. In other words, although a particular estimate represents a confidence interval $p \pm w$ for some α% of confidence, the estimate is simply set to the sample mean, p. This is done since the ns used in the algorithms are expected to be large, i.e., $2\sqrt{p(1-p)/n}$ will likely be insignificant.

4 The Revealing Ability Metric

Section 4 presents formulae quantifying (1) the revealing abilities of program locations and (2) the revealing ability of programs. Parameters to these formulae are the probability estimates produced by propagation, infection, and execution analysis. Recall that revealing ability is a tool for making predictions—revealing ability uses estimates from previous

observed computational behavior to suggest future computational behavior. Additionally, Section 4.2 describes how a tester might apply the revealing abilities of program locations as the foundation for a white-box testing tool during module testing.

4.1 The Revealing Ability Metrics

Applying propagation, infection, and execution analysis to a large program (of say more than 100 program locations) produces a voluminous number of probability estimates. Although each distinct probability estimate is informative, the number of probability estimates will likely be overwhelming, so there must be a way to collapse that information into fewer individual parts. The revealing ability metric does just this; it takes in each probability estimate for a program location and produces a single revealing ability for the program location. Revealing ability, whether for a program or program location is in [0,1], and there are a continuum of revealing abilities in [0,1].

4.1.1 Program Location Revealing Ability

To get a feeling for how propagation, infection, and execution estimates are related to whether faults will be revealed, consider a program location that is infrequently executed given a particular input distribution. Such a program location could more easily protect a fault from detection during random testing because the program location is rarely given an opportunity to affect the program states. Further consider a program location that can easily sustain injected syntactic mutants or injected program state mutants and produce little or no discernible effect in subsequent program states concerning the injection of such semantic mutants. Small infection and propagation estimates also suggest that this is a program location that could more easily protect a fault from detection during random testing. In this metric, program locations that receive small propagation, infection, and execution estimates will be duly recognized as having a greater potential for allowing faults to remain undetected during random testing and will be assigned lower revealing abilities.

ξ_l represents the revealing ability of program location l. The revealing ability of program location l is:

$$\xi_l = (\frac{w_1}{n_l} \cdot \sum_{k=1}^{n_l} \hat{\psi}_{l,a_k}) + (\frac{w_2}{m_l} \cdot \sum_{i=1}^{m_l} \hat{\lambda}_{l,p_i}) + (w_3 \cdot \hat{\varepsilon}_l), \quad (1)$$

where w_1, w_2, and w_3 are weights associated with the importance of the three sets of estimates; n_l is the number of live variables at program location l; and m_l is the number of syntactic mutants injected at program location l. Values for the weights are arbitrary, however $w_1 + w_2 + w_3 = 1.0$; this keeps revealing abilities in [0,1].

For program location l, ξ_l is a linear combination of the mean of the infection estimates, the mean of the propagation estimates, and the execution estimate. The weights are provided in equation **??** to provide flexibility when considering the importance of the three techniques. For instance, if it is felt that the information provided by execution analysis is the most important information provided by the three techniques in predicting where faults could remain undetected, then w_3 can be fixed at greater than 1/3. By introducing weights into equation **??**, the user of the metric can either consider these three sets of estimates as equally important, when $w_1 = w_2 = w_3 = 1/3$, or consider biasing a particular weight.

With the revealing ability of each program location available, a tester receives the following benefits:

1. *Where to get the most value from limited testing resources:*
   In general, high revealing ability program locations require less random testing than low revealing ability program locations for the same confidence that the program location is not protecting a fault from detection [**?**, **?**]. By identifying high revealing ability program locations, testing resources can be shifted by the tester to the more troublesome low revealing ability program locations. The revealing abilities of program locations can be used as a white-box testing tool for where to emphasize testing resources. This is discussed in Section 4.2.

2. *Which program locations should receive some validation technique other than random testing:*
   Quantification of the metric may reveal an extremely low revealing ability at a particular program location or a section of program locations, thereby pinpointing program locations or sections of program locations where unreasonable amounts of random testing under the assumed input distribution are needed. At these program locations, alternate validation techniques such as verification, code review [**?**], or exhaustive testing should be considered.

4.1.2 Program Revealing Ability

The revealing ability of a program is derived from the individual program locations' ξ_ls that comprise it. The revealing ability of a program P is denoted by Ξ_P. Ξ_P, for a program P containing N program locations, is:

$$\Xi_P = \frac{1}{N} \cdot \sum_{l=1}^{N} \xi_l. \tag{2}$$

With the revealing ability of a program, a tester receives the following benefit:

1. *Whether the software or sections of the software should be rewritten:*
   Revealing ability may be used as a tool for deciding whether safety-critical software [**?**] has been sufficiently verified. If a section of a safety-critical program contains many program locations of low revealing ability, then the section should be considered for rejection, since enormous quantities of random testing will be required to verify a sufficient level of confidence. In order to avoid rejecting software because of a low revealing ability, [**?**, **?**, **?**] presents preliminary ideas on how modules can be designed to enable faults to be more easily found during random testing; the technique is based on the suggested ability of a module to protect faults from detection *before the module is written*. This is a static technique that is based on the specification of a module. The goal of this design technique is to produce modules that either have high revealing ability or low revealing ability. Modules that are assigned a low revealing ability can possibly be verified if they are minimized syntactically, and those that are assigned a high revealing ability can be tested with **a priori** knowledge that they are less likely to protect faults from detection.

4.1.3 Example

The revealing ability metric is now demonstrated on several trivial programs. Consider two programs that implement function f, where $f(x) = x+1000$. In these examples, it is assumed that the initial value for variable **x** has already been read in. The first program, P_1, is:

$\{l_1\}$ **x := x + 1000;**

The second program, P_2, is:

$\{k_1\}$ **x := x + 1;**
$\{k_2\}$ **x := x + 1;**
•
•
•
$\{k_{1000}\}$ **x := x + 1;**

Although P_2 is a ridiculous implementation of f, it is correct. For P_1, with only one program location l_1, ξ_{l_1} and Ξ_{P_1} are near 1.0, since

1. The single propagation estimate is 1.0 (there is only one propagation estimate because there is only one live variable **x**),

2. The execution estimate is 1.0, and

3. The infection estimates will be near 1.0 (because l_1 implements a one-to-one function).

An example of a syntactic mutant for program location l_1 that does not have a 1.0 infection estimate is **x := x * 1000**, assuming $x = 1000/999$ is in the input domain of P_1. For P_2, its revealing ability is also near 1.0, because similar arguments can be made for each program location $(k_1, k_2, ..., k_{1000})$ in P_2 that were made for program location l_1. So P_1 and P_2 have approximately the same revealing ability.

Static software metrics that are based on the number of operators or number of lines of code might consider programs such as P_2 as more complex than programs such as P_1, which in terms of readability or debugging is true. However if the ability to measure the resolve of a program to reveal faults is the metric's objective, the metric should incorporate some notion of the ability of the program's locations to be executed, create program state errors, and propagate program state errors to the output. When these criteria are used for determining testing complexity, it is found that P_1 and P_2 are equally complex. This more semantic view of software is the intuition behind the revealing ability metric, and as this example shows, greater amounts of syntax does not necessarily mean greater semantic testing complexity. In general, however, complex syntax does suggest greater syntactic testing complexity. Revealing ability just provides an additional viewpoint from which to discuss testing complexity, i.e., a semantic viewpoint.

For another example that demonstrates this, consider a function g, where $g(x) = x \bmod 2$, and one obvious implementation, P_3, of g:

$\{m_1\}$ **x := x mod 2;**

The propagation estimate and execution estimate for program location m_1 are both 1.0 since there is only one location, however the infection estimates for this location may be less than 1.0 for certain syntactic mutants. For instance, program location m_1 could be replaced with the syntactic mutants **x:= x mod 4** or **x:= x mod 1**, and if the input distribution to P_3 is uniformly distributed over even and odd values of x, then less than 100% of the inputs to this erroneous implementation would produce a program state error, which in this case would also be a program failure. So the revealing ability of P_3 would be less than 1.0 if these two syntactic mutants were used during infection analysis, because the syntactic mutants **x:= x mod 4** and **x:= x mod 1** have infection estimates that are less than 1.0 for uniformly distributed values of x.

These examples show that P_3 is believed to more easily protect faults from detection than P_1 and P_2. An important difference in these programs is that P_3 implements a many-to-two function, and P_1 and P_2 implement a one-to-one function. A conjecture presented

in [?, ?, ?] argues that a program's ability to protect faults from detection during random testing is partially related to the ratio between the cardinality of the domain to the cardinality of the range of the function that the program implements. Experience has shown that this conjecture generally holds for trivial examples such as P_1, P_2, and P_3. Whether the conjecture holds for non-trivial examples is under analysis.

4.2 White-Box Testing Tool

The revealing ability of a program location suggests the program location's effect on the program's computational behavior during testing. The lower the revealing ability of a program location, the lesser the believed effect. With the revealing abilities of program locations, those program locations can be isolated that have a lesser suggested effect and concentrate testing efforts there. If a region of a program contains many program locations with low revealing abilities, then possibly such a region should be redesigned or rewritten.

A white-box testing strategy for determining where to concentrate testing resources in a program or deciding when to apply other validation techniques can be based on the revealing abilities of the program locations. If a particular ξ_l is large, say greater than some constant α, then program location l may not require additional amounts of testing. If a particular ξ_l is small, say less than α, then possibly program location l should receive additional analysis, either through additional testing or other validation techniques. (α is an arbitrary threshold; determining how to find the best threshold has not yet been investigated.)

Since we rarely would ever test a single program location, a more practical way of using the ξ_ls in a white-box testing capacity would be to determine the revealing ability of a module. That is, apply equation 2 to a module of N program locations instead of to a complete program. If a particular module receives an extremely low revealing ability, then concentrating validation resources on such a module would be prudent.

5 Concluding Remarks

The paper presents a dynamic metric termed revealing ability—the revealing ability metric suggests whether the increase in confidence in the software's correctness produced from random testing without observing failures can be justified. Revealing ability is a semantic metric, as opposed to most other metrics that are syntactic. The importance in this work is not in the proposed metric itself, but rather in exposing the notion of semantic software testing metrics.

The lower the revealing ability, the greater the suggested difficulty that will be incurred

from undetected faults during random testing. A low revealing ability is not an indictment against software testing, but instead identifies a potential limitation of random testing when the goal of such testing is to reveal faults. The revealing ability metric suggests that testing a program with high revealing ability can produce extremely high confidence in the software's correctness. For program's with low revealing ability, questions about whether program inputs have the capacity to reveal faults assuming faults were to exist are raised.

Not only can revealing ability be used to judge the effectiveness of random testing, but it can be used as the foundation for a white-box testing tool for determining where testing resources are most needed. Revealing ability can be used to balance testing in such a manner as to not over test in high revealing ability regions and not under test in low revealing ability regions.

Although revealing ability may always seem beneficial, it can create problems. It may happen that a perfectly *correct* program has an extremely low revealing ability. This situation might cause strong action to be taken against the program, even to the point of rejecting the program. Remember that revealing ability and correctness are not directly related; revealing ability only suggests the likelihood that faults will be revealed during random testing assuming faults exist. This does not say whether the software is correct. This unfortunate situation appears to be unremediable, however, I contend that the advantages to testing provided by revealing ability to most programs outweigh the disadvantages caused to correct programs with low revealing abilities.