

# Examining Fault-Tolerance Using Unlikely Inputs: Turning the Test Distribution Up-Side Down

Jeffrey M. Voas (jmvoas@RSTcorp.com)  
Reliable Software Technologies Corp.  
Suite 250, 21515 Ridgetop Circle  
Sterling, VA 20166

Keith W. Miller (miller@eagle.sangamon.edu)  
Department of Computer Science  
Sangamon State University  
Springfield, IL 62794

## Abstract

*This paper turns the concept of input distributions on its head to exploit “inverse input distributions.” Although such distributions are not true mathematical inverse functions, they capture the intuitive property of members that have high frequencies in the original distribution have low frequencies in the “inverse” distribution, and vice versa. These new distributions have uses in testing for reliability estimates, and more importantly, fault-tolerance analysis.*

## 1 Background

Good engineering practice in software development is obviously a necessity on the most intuitive grounds: we cannot expect to get away with haphazard construction of the most complex objects in human history. Quantitative fault-tolerant measurement is also necessary because *direct* measurement of adequate software quality is impractical, and will probably always remain so. This paper presents an *indirect* fault-tolerance measurement technique that is practical.

Software engineers that are concerned with software reliability estimation traditionally use the likely distribution of inputs. Such a distribution is termed an *operational profile* [7, 5]. When we talk about an operational distribution here, we are only concerned with the data that the program reads in, not the platform or external environment that encompasses the software.

In this paper, we turn the concept of operational distributions on its head to exploit these distributions in a new way. We examine “inverse operational distributions.” Although such distributions are not true mathematical inverse functions, they capture the intuitive property of elements that occur frequently in the original operational distribution,  $Q$ , occur infrequently in the “inverse operational distribution,”  $\bar{Q}$ . This new distribution has uses in testing, testing for reliability

estimates, and most importantly, fault-tolerance analysis.

Our meaning of the term *software fault tolerance* is different than the traditional hardware definition.<sup>1</sup> Our research considers a single program to be fault-tolerant if and only if:

1. the program is able to compute the correct result even if the program itself suffers from incorrect logic, *and*
2. the program, whether correct or incorrect, is able to compute the correct result even if the program itself receives *corrupted* incoming data during execution.

Programs with high degrees of these characteristics have been termed *self-correcting*.

Widely accepted software engineering design practices have argued for *robustness* and *graceful degradation* whenever a system gets into an undesirable state [8]. Software fault tolerance is a related concept, yet distinct. The distinction between robustness and fault tolerance rests on whether the undesirable state is “expected” or “unexpected.” Robustness deals primarily with problems that are expected to occur and must be protected against. In contrast, fault tolerance primarily deals with problems that are unexpected, yet they too must be protected against. For example, if we are reading in an integer that will be used in a division operation, a robust design will ensure that the division operation is not applied if the integer is zero. A fault-tolerant design accounts for unanticipated possibilities, e.g., if the integer is corrupted, a fault tolerant design might freeze the state of the program and not compute the division operation (which is equivalent

---

<sup>1</sup>In fact, even the term “fault-tolerance” when applied to software usually suggests the application of multiple versions, multiple processors, or recovery block schemes, but that is not our intent here; we are talking about fault-tolerance in the purest sense, where *any* anomaly that is manifested during execution can be thwarted.

to an integer divide-by-1), or it might require that the integer be reread. Here, we are interested in assessing fault-tolerance, which can be a side-benefit of robust design practices.

## 1.1 Extended Propagation Analysis

Our fault-tolerance assessment method that uses inverted operational distributions relies heavily on the power of a technique termed “extended propagation analysis” (EPA) [4].<sup>2</sup> Since EPA is pivotal to our fault-tolerance assessment method, we will now describe EPA.

EPA collects *dynamic* information concerning which output variables are affected by a data state value that is “somehow” altered. This analysis is related in purpose to *static* fault-tree analysis. EPA differs from conventional fault-tree analysis because (1) the backwards trace is made after EPA executes the program, not from static control-flow analysis, and (2) EPA concentrates on data state propagation, not software faults. Fault-tree analysis is concerned with the combinations of component failures that can lead to system failures, and therefore works backwards from the system to its components. EPA isolates program regions: if these regions create certain types of data state errors, then we predict that certain types of software failure will result. EPA is also related to *failure modes and effects analysis* (FEMA) [2], which provides for the consideration of different types of failures by working forward from the components to the system. FEMA postulates an anomalous event, and studies whether the event can propagate to the output space, which is conceptually similar to the processes of EPA.

For software to be fault-tolerant, there are two classes of “problems” that must be protected against: software faults and hardware failures (or what are sometimes termed as “faults”), i.e., erroneous incoming data to the software. EPA simulates both classes, and thus we know the impact on system output if a hardware sensor were to malfunction or if the software itself were to malfunction (with respect to the specification). This allows us to assert with confidence exactly what the net impact is on critical output variables based on the fault classes (hardware and software) that we simulate, and whether this “net impact” turns out to be a possible catastrophe.

<sup>2</sup>EPA is a spin-off of Voas’s Sensitivity Analysis technique [11]; the main differences are that EPA is only concerned with the propagation condition, and EPA is also concerned with incoming hardware failures as opposed to Sensitivity Analysis that is only concerned with resident program faults. Also, EPA differentiates classes of failure; Sensitivity Analysis does not.

### 1.1.1 The EPA Theoretical Model

A *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion between locations (statements).<sup>3</sup> We refer to a data state error as an *infection*, and use these two terms interchangeably. If a data state error exists, the data state and variable with the incorrect value at that point are termed *infected*. A data state may have more than one infected variable. *Propagation* of a data state error occurs when a data state error affects the output.

Let  $S$  denote a specification,  $P$  denote an implementation of  $S$ ,  $x$  denote a program input,  $\Delta$  denote the set of all possible inputs to  $P$ ,  $Q$  denote the probability distribution of  $\Delta$ ,  $l$  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  $x$ . And let  $\mathcal{A}_{lPx}$  represent the data state produced after executing location  $l$  on the  $i^{\text{th}}$  execution from input  $x$ .

It is important to group data states into sets with similar properties. For instance, assume that location  $l$  is executed  $n_{xl}$  times by input  $x$ . Now consider all of the data states that are created by this input immediately before  $l$  is executed or immediately after  $l$  is executed. The following set allows us to do so:

$$\mathcal{A}_{lPx} = \{\mathcal{A}_{lP_i x} \mid 0 \leq i \leq n_{xl}\}$$

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

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

A *simulated infection* is a modified value forced into the value of some variable (that already had a different value) in a data state. As we have already stated,  $\mathcal{A}_{lP_i x}$  denotes the data state created after the  $i^{\text{th}}$  iteration of location  $l$  on input  $x$ ;  $\hat{\mathcal{A}}_{lP_i x}$  denotes this same data state after a simulated infection is injected into  $\mathcal{A}_{lP_i x}$ . A simulated infection usually affects a single live variable.

It is important at this point to explain the relationship between simulated infections and the potentially disastrous states that a system can get into that can lead to a catastrophic output. When a system gets into a “bad state” during execution, the next event that we would like to see occur is recovery from that state back to an acceptable state. Simulated infections are the mechanisms that are employed in EPA to allow observation of the impact of different classes of

<sup>3</sup>Admittedly, this is tenuous, since for any specification, there is an infinite number of correct programs that implement that specification, and therefore for each statement in a program version there must be an assertion if we are to expose data state errors, which as a practical matter will never happen.

“bad states.” Simulated infections mimic the effect of both programmer faults and hardware failures (coming into a system). Simulated infections are created by perturbation functions. The process of injecting a simulated infection into an executing program 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 consider that program  $P$  has a fixed set of output variables:  $\{v_1, v_2, v_3, \dots, v_n\}$ . An *affected variable* is an output variable whose value differs after a simulated infection is forced into the program and the program execution is resumed and termination occurs. For instance, if after a simulated infection is forced into the program, program execution is resumed, termination occurs, and output variable  $v_3$  contains a different value, then  $v_3$  is an affected variable.

The following EPA algorithm creates sets of affected variables that occur after some variable  $a$  is perturbed on all iterations at location  $l$ . Because we are interested in the program’s fault tolerance under circumstances with both software faults and incoming data corruptions, this algorithm will be applied to every program variable, including input variables.

**Algorithm 1:**

1. Set **k** to 0.
2. Set **variable\_set** =  $\emptyset$ .
3. Increment **k**.
4. Randomly select an input  $x$  according to  $Q$ , and if  $P$  halts on  $x$  in a fixed period of time, find the corresponding  $\mathcal{A}_{lPx}$  in  $\alpha_{lP\Delta}$ . Set  $\mathcal{Z}$  to  $\mathcal{A}_{lP1x}$ .
5. 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  $\mathcal{Z}$ . If  $l$  is executed more than once for  $x$ , i.e.,  $\mathcal{A}_{lP2x}, \dots, \mathcal{A}_{lPmx}$ , alter  $a$  in each  $\mathcal{A}_{lPix}$ ,  $2 \leq i \leq m$ .
6. For each output variable that contains a different value (after comparing  $P$ ’s output using  $\check{\mathcal{Z}}$  to the output  $P$  regularly produces), add it as a member to the set **variable\_set**.
7. Set  $\Pi_{alPQk} = \mathbf{variable\_set}$ . ( $\Pi_{alPQk}$  represents the set of output variables

that have different values given execution of  $P$  occurs with  $\check{\mathcal{A}}_{lPx}$ .  $\Pi_{alPQk}$  is the empty set if no output variables are affected by the injection of  $\check{\mathcal{A}}_{lPx}$ .)

8. Repeat steps 2-7  $n$  times.

There will be instances, however, were we are not necessarily concerned with whether variable corruption occurred, but more specifically whether a particular output event occurred, which we will denote by predicate  $PRED$ . To determine this, we do not need to run the unperturbed version of program  $P$ .  $PRED$  will represent a predicate expression that relates specific variables to values ranges or combinations of variables and ranges. Also,  $PRED$  may contain certain restrictions on the input that was used during an execution. The following algorithm provides this information; it determines the proportion of outputs that satisfy  $PRED$ :

**Algorithm 2:**

1. Set **count** to 0.
2. Randomly select an input  $x$  according to  $Q$ , and if  $P$  halts on  $x$  in a fixed period of time, find the corresponding  $\mathcal{A}_{lPx}$  in  $\alpha_{lP\Delta}$ . Set  $\mathcal{Z}$  to  $\mathcal{A}_{lP1x}$ .
3. Alter the sampled value of variable  $a$  found in  $\mathcal{Z}$  creating  $\check{\mathcal{Z}}$ , and execute the succeeding code on  $\check{\mathcal{Z}}$ . If  $l$  is executed more than once for  $x$ , i.e.,  $\mathcal{A}_{lP2x}, \dots, \mathcal{A}_{lPmx}$ , alter  $a$  in each  $\mathcal{A}_{lPix}$ ,  $2 \leq i \leq m$ .
4. If the output satisfies  $PRED$ , increment **count**.
5. Repeat steps 2-4  $n$  times.
6. Divide **count** by  $n$  yielding  $\hat{\psi}_{alPQ}$ ; ( $1 - \hat{\psi}_{alPQ}$  is the degree of *fault-tolerance*).

Step 5 of Algorithm 1 and similarly Step 3 of Algorithm 2 are critical to the value of the information produced by these algorithms; the alterations of  $a$  must be reflective of either a hardware failure class or a class of programmer faults. For example, Underwriter’s Laboratory’s standard, “Safety-Related Software,” [10] defines the following classes of events that would need to be simulated in Step 5 to be used for their standard:

“These requirements [UL1998] address risks that may occur as a result of faults caused by software errors, such as the following: a) design errors such as incorrect algorithms or

interfaces b) Coding errors, including syntax, incorrect signs, endless loops, and the like; c) Timing errors that can cause program execution to occur prematurely or late; d) Induced errors caused by hardware failure; e) Latent errors that are not detectable until a given set of conditions occur;.....”

Step 5 of Algorithm 1 (and similarly Step 3 of Algorithm 2) can demonstrate what types of outputs are produced by the class of anomalies mentioned in b), c), and d) of UL1998. We will later show how using inverted distributions in Step 4 of Algorithm 1 and Step 2 of Algorithm 2 can demonstrate the risks mentioned in e) (See Section 1.4). Also, NASA’s new interim software safety standard (due to be approved in 1995) requires a demonstration of software fault-tolerance to problems in timing and hardware failure sensitivities [6]. These anomalies can also be simulated by these algorithms.

Algorithm 1 produces the sets:  $\Pi_{alPQ1}, \Pi_{alPQ2}, \dots, \Pi_{alPQn}$ . We then create a set of these sets:

$$\{\Pi_{alPQk} \mid 1 \leq k \leq n\}$$

This set now represents all combinations of output variables that experienced different values when  $a$  was perturbed at  $l$ . Note that Algorithms 1 and 2 can use any distribution that is derived from  $Q$ , which will be important later.

## 1.2 Using Extended Propagation Analysis To Detect “Dangerous” (Unsafe) Locations

We now take the information provided by Algorithm 1 and produce the set of locations from which a particular type of software failure could result. Recall that the goal here is to identify where in the code specific catastrophic failures could originate. Although we will not show it here, a similar analysis could be performed using the results of Algorithm 2 instead of Algorithm 1.

Let  $\mathcal{V}_P$  represent a set of sets. Each member of  $\mathcal{V}_P$  contains either a single output variable or a combination of output variables of program  $P$ . Each member of  $\mathcal{V}_P$  represents one type of software failure of  $P$ . For instance, if  $\mathcal{V}_P = \{\{v_1\}, \{v_2, v_3, v_4\}\}$ , then we have identified 2 types of software failure: the first type occurs when the single output variable  $v_1$  is incorrect, and the second type occurs when the output variables  $v_2, v_3$ , and  $v_4$  are all incorrect. If we apply Algorithm 1 and

$$(\exists k \mid 1 \leq k \leq n)(v_1 \in \Pi_{alPQk})$$

we predict that if the value of variable  $a$  at location  $l$  is incorrect, output variable  $v_1$  will be incorrect. Thus if location  $l$  is incorrect and this “incorrectness” affects the value of  $a$ , the first failure type is predicted to occur. If

$$(\exists k \mid 1 \leq k \leq n)(v_2 \in \Pi_{alPQk}) \wedge (v_3 \in \Pi_{alPQk}) \wedge (v_4 \in \Pi_{alPQk})$$

we predict that the second type of failure will occur if location  $l$  causes variable  $a$  to be incorrect. Performing this analysis isolates locations that we predict can cause a class of software failure defined in  $\mathcal{V}_P$ . Dynamically, this reveals that there is a location  $l$  that is not fault-tolerant for the classes of failure in  $\mathcal{V}_P$ .

This has a direct application to safety critical software. Let  $\mathcal{C}_P$  represent a set of sets that is similar to  $\mathcal{V}_P$  above. Each internal set in  $\mathcal{C}_P$  contains either a single output variable or a combination of output variables of program  $P$ .<sup>4</sup> If either the single output variable or combination of output variables are ever corrupted, a catastrophic event of the system that  $P$  controls will result. When we apply Algorithm 1 and

$$(\exists k \mid 1 \leq k \leq n)[(\exists \gamma \mid \gamma \in \mathcal{C}_P) (\gamma \subseteq \Pi_{alPQk})] \quad (1)$$

we predict that if the value of variable  $a$  at location  $l$  is corrupted, critical software failure is possible. The  $l$ s, where Equation 1 is true, are code regions that warrant concern during the operational phase, since these regions have the potential to propagate unsafe data state errors into critical software failures. This is the set of locations that are candidates for additional fault-tolerant mechanisms.

## 1.3 Example

Let’s return to our previous example to demonstrate this technique. Recall that if

$$(\exists k \mid 1 \leq k \leq n)(v_2 \in \Pi_{alPQk}) \wedge (v_3 \in \Pi_{alPQk}) \wedge (v_4 \in \Pi_{alPQk})$$

were true, we would predict that the second type of failure will occur if location  $l$  causes variable  $a$  to be corrupted. Suppose that we wish to be more precise and we decide that it is not enough to cause alarm when  $v_2, v_3$ , and  $v_4$  are corrupt, but instead it is of greater concern when their output values are such that  $v_2 > 10$ ,  $v_3 = 360$ , and  $0 \leq v_4 < 3.12345$ . We can

<sup>4</sup> $\mathcal{C}_P$  is determined during hazard analysis in the requirements phase and is directly related to system safety requirements.

tighten our condition for warning about a catastrophic event originating from location  $l$  and variable  $a$  to

$$(v_2 > 10) \wedge (v_3 = 360) \wedge (0 \leq v_4 < 3.12345)$$

from  $(\exists k \mid 1 \leq k \leq n)(v_2 \in \Pi_{alPQk}) \wedge (v_3 \in \Pi_{alPQk}) \wedge (v_4 \in \Pi_{alPQk})$ . As you can see, care is required when we define catastrophic events. The less ambiguous the description, the more precise the result.<sup>5</sup> The more limited the description, the fewer situations will be marked as potentially disastrous. We will simplify the determination of whether a catastrophic event has occurred to the following classes of events:

1. **Unperturbed, *PRED* Violated** To reveal this information, it will be necessary for the user to specify that they desire for the unperturbed version to be tested for whether it's output violates *PRED*. In this situation, the version without fault injection produces an output that violates *PRED*, and the user must be informed. Warning of a catastrophic event occurring from the *unperturbed* version requires a simple test of the programs output against *PRED*. testing the output against *PRED*.
2. **Unperturbed, No Violation** Nothing to report here.
3. **Perturbed, *PRED* Violated** This will be considered as a catastrophic failure; observing this event does not require executing the unperturbed version.
4. **Perturbed, Corrupted, No *PRED* Specified** This will be counted as a catastrophic failure; determining this requires executing the unperturbed version.

We can generalize these two methods of warning of a catastrophic event occurring from the *perturbed* version and originating at location  $l$  to:

$$(\exists k \mid 1 \leq k \leq n)[(\exists \gamma \mid \gamma \in \mathcal{C}_P) (\gamma \subseteq \Pi_{alPQk})] \vee \text{PRED} \quad (2)$$

## 1.4 Inverted Distributions: Definitions

We have thoroughly described the technique that we will use to assess fault-tolerance, EPA. Now it is time to describe the method that we will use for assessing the fault-tolerance of the ultra-reliable region, which is the main contribution of this paper.

<sup>5</sup>Algorithm 2 is designed to be more precise than Algorithm 1 by allowing for *PRED*. All catastrophic events should be defined by someone familiar with the complete system.

The *input space*,  $\Delta$ , is the set of all legal and possible (meaning a greater than 0 probability) input values represented by the specification's domain. For each member of the input space, there is a likelihood that the member is selected during testing or use. We term those members that are likely to be selected during testing as members of the *probable input subdomain*, and the rest are members of the *improbable input subdomain*. The union of these two subdomains is simply  $\Delta$ . The input distribution for a piece of software is a probability density function that assigns to each possible input test case  $i$  a probability that  $i$  will be selected on a randomly selected execution of the software in a certain environment. There are difficulties, both theoretical and practical, in obtaining such a distribution [5], but here we will assume that an estimated input distribution is available. One may always be estimated, though its accuracy may be suspect. All true probability density functions  $Q$  have the property that the sum of  $Q(i)$  over all legal  $i$  equals 1. Each  $Q(i)$  is a probability, and thus is between 0 and 1 inclusive. If a particular  $Q(i)$  equals 0, then  $i$  is not a member of the input space as we previously defined it. If a particular  $Q(i)$  equals 1,  $i$  is the *only* member of the input space.

Here, we examine how to exploit a new distribution which in an informal sense “inverts” the original input distribution. Intuitively, we want the new distribution, call it  $\bar{Q}$ , to assign a large probability to elements that had a small probability in  $Q$ , and to assign a small probability to elements that had a large probability in  $Q$ . Building an “inverse” distribution algorithmically requires several subjective decisions. First, what does it mean when  $Q(i) = 0$  in the original distribution? If it means that  $i$  never occurs as an input to the software, then we should disregard  $i$  in the new distribution as well; if  $Q(i) \approx 0$  because it occurs only very rarely, or because the developer only expects it to be used rarely, then we want the inversion to give  $i$  a relatively high probability of selection. Here, we will arbitrarily select the first decision:  $\bar{Q}(i)$  will be assigned 0 for each  $i$  such that  $Q(i) = 0$ .

Next, we must decide how to obtain a new distribution  $\bar{Q}$  that captures the intuition of an inverse. A true inverse operation would have the property that  $inverse(inverse(Q)) = Q$ , but the peculiar properties of a probability density function make this difficult to attain. Instead, we offer the following constructive definition of the “inverse,”  $\bar{Q}$ :

### Algorithm 3:

1. Let  $N$  be the number of different legal inputs. Let  $M = 1/N$ , the mean of the

probability density distribution over  $N$  possible inputs.

2. For each element  $i$ , let  $g'(i) = 2 * M - Q(i)$ .
3. Find the minimum  $g'(i)$ ,  $m$ . If  $m \geq 0$ ,  $g'(i)$  is  $\bar{Q}$ . Otherwise, proceed to step 4.
4. When  $m < 0$ , let  $g''(i) = (g'(i) + \mathbf{abs}(m))/(1 + N * (\mathbf{abs}(\mathbf{m})))$ . Then  $g''$  is  $\bar{Q}$ .

An alternative  $\bar{Q}$  could be formed by setting all negatives in  $g'$  to zero, and then re-normalizing. However, we rejected this because it loses the intuitive idea of retaining the inverse shape of the distribution.

As an example, Figure 1a shows an input distribution for a single integer variable from  $-5$  to  $+4$ . The height of the bar represents the probability that the associated input (an integer in Figure 1a) will be chosen in a random selection from the input. Any theoretically possible inputs that do not appear in the graph have a zero probability of selection.

In Figure 1b, a line indicates the average probability of selection for the legal inputs. This average probability line (or plane) defines the uniform random distribution over legal inputs. An input that has a probability above this line (or plane) we call a “likely” input; an input below this line (or plane) is “unlikely.” In order to invert the distribution in Figure 1a, we reflect the distribution about the uniform distribution line (or plane). Figure 1c shows the resulting  $\bar{Q}$ .

In Figure 1c, the inverse of  $\bar{Q}$  will return  $Q$ . However, not all input distributions are so nicely behaved. Figure 1d shows an input distribution that, when reflected about the uniform distribution, produce Figure 1e resulting in negative probabilities, which are undefined, as visible in Figure 1f. This situation requires a more elaborate transformation, which makes the inversion non-standard. We’ll describe this transformation in two steps: first, we shift the graph in Figure 1f up by a constant equal to the absolute value of the smallest negative value in the  $y$  direction, yielding the graph in Figure 1g. This graph is non-negative, but the  $y$  value no longer adds to one, so our second step requires normalization, i.e., dividing each  $y$  value value by the sum of all the  $y$  values. The resulting graph is shown in Figure 1h.

## 1.5 $\bar{Q}$ and Traditional Testing

Performing EPA with  $\bar{Q}$  identifies code locations that are unlikely to cause failures when rare inputs are seen. These locations can either: (1) contain design logic flaws or (2) be input statements that read

corrupted external data that is coming into the software from the system that the software is controlling. If either type of event actually occurs, and

$$(\exists k \mid 1 \leq k \leq n)[(\exists \gamma \mid \gamma \in \mathcal{C}_P) (\gamma \subseteq \Pi_{alP\bar{Q}k})], \quad (3)$$

is true, then we know that in  $P$ ’s life-time, there is the possibility that  $P$  will fail catastrophically when rare inputs are selected if an anomalous event corrupts the state at location  $l$ .

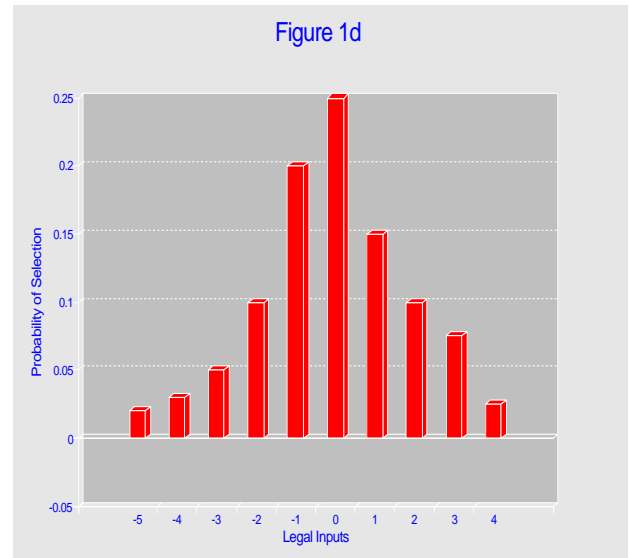
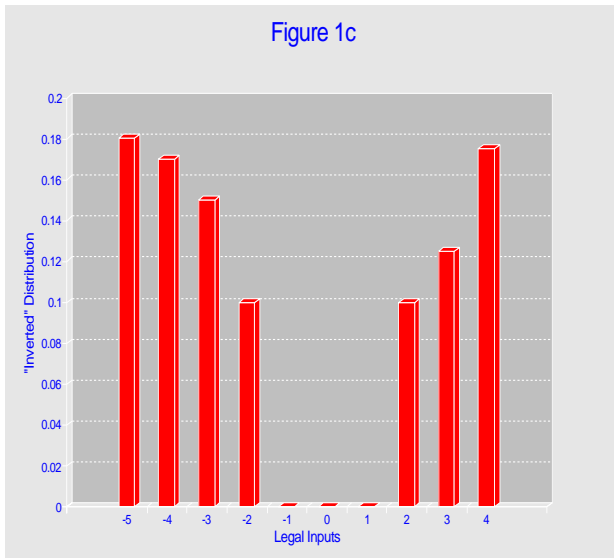
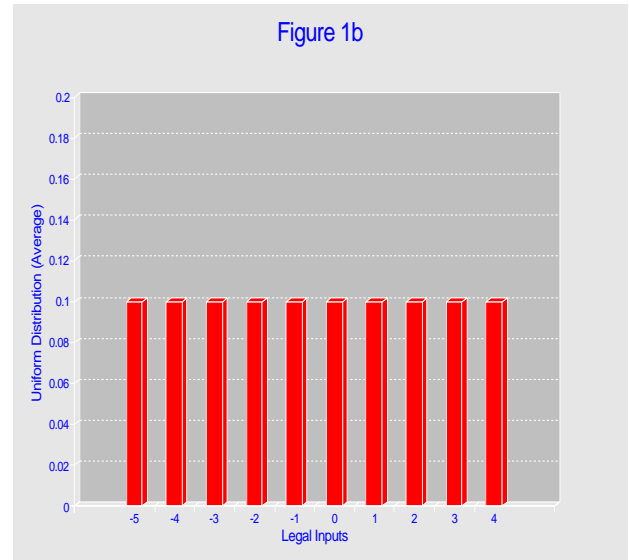
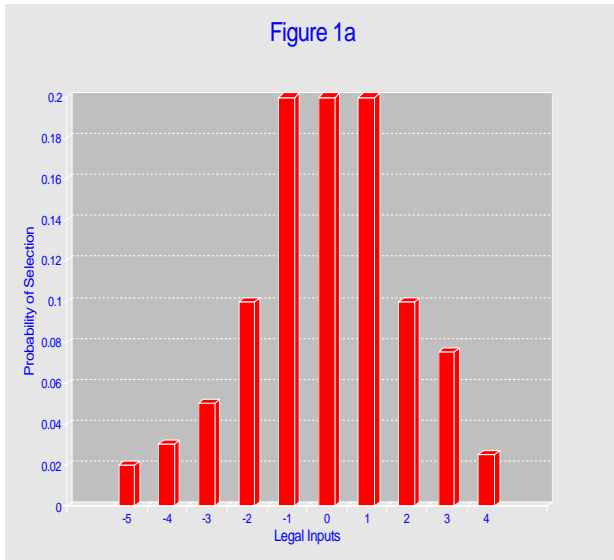
To illustrate the importance of EPA using both  $Q$  and  $\bar{Q}$ , we consider four extreme cases:

1. A particular location (or module or program) is insensitive under *both* distributions.<sup>6</sup> This location is unlikely to reveal any faults during testing. It is therefore a prime candidate for formal methods or other non-empirical verification schemes. A fault in this location would be very difficult to find during testing with the original distribution, so any remaining faults are potential surprises after release.
2. A particular location is sensitive under  $Q$  but insensitive under  $\bar{Q}$ . Testing using  $Q$  will likely reveal faults. But during operation, we have a confidence that if the code experiences any problems similar to the “mimicked” faults used during EPA, the code is self-correcting enough to not fail.
3. A particular location is insensitive using  $Q$ , but sensitive using  $\bar{Q}$ . This location will benefit from testing using  $\bar{Q}$ .
4. A particular location is sensitive under both distributions. Testing using either distribution is likely to reveal any faults.

When we are interested in whether or not testing will discover faults, high fault-tolerance can be discouraging. But fault tolerance requires that software be robust and able to recover from unexpected problems, as well as not “reveal” problems if they exist. That is, the software should be able to somehow recover from a software fault or corrupted input and produce correct (or at least acceptable) output. During testing, we want faults revealed; during fault tolerant execution, we want faults “covered up.”

If testing is based primarily on a particular input distribution, then inputs in a low probability bin of that distribution are less likely to trigger failures during execution. The software segments exercised by

<sup>6</sup>High sensitivity implies low fault tolerance, and insensitivity implies high fault tolerance [4].



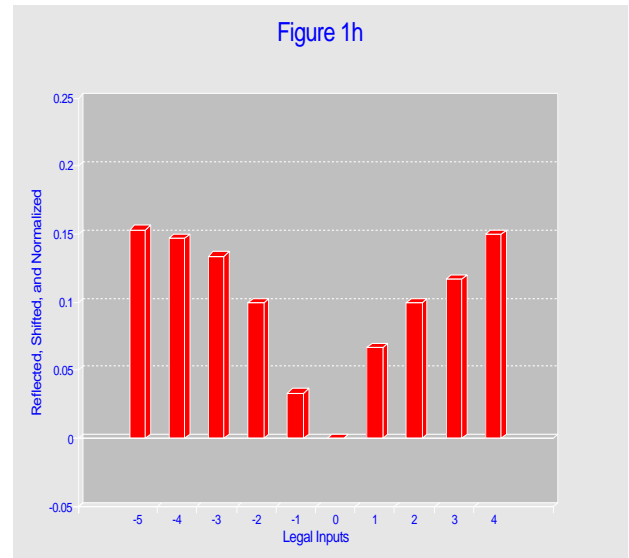
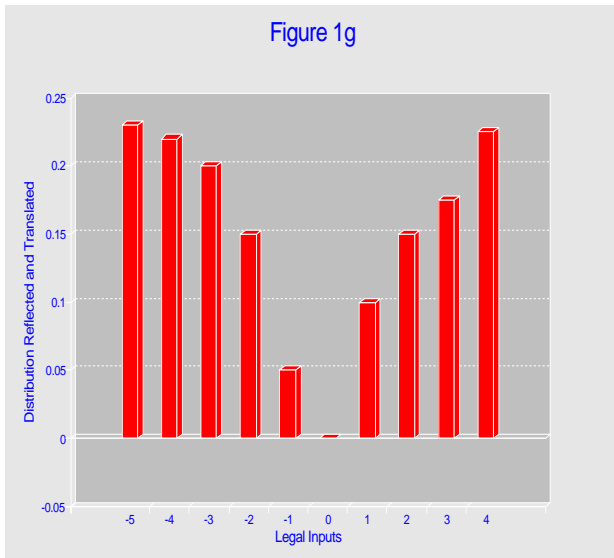
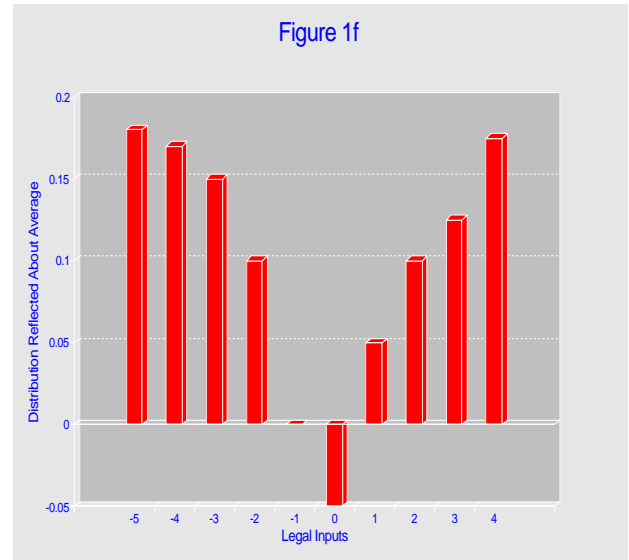
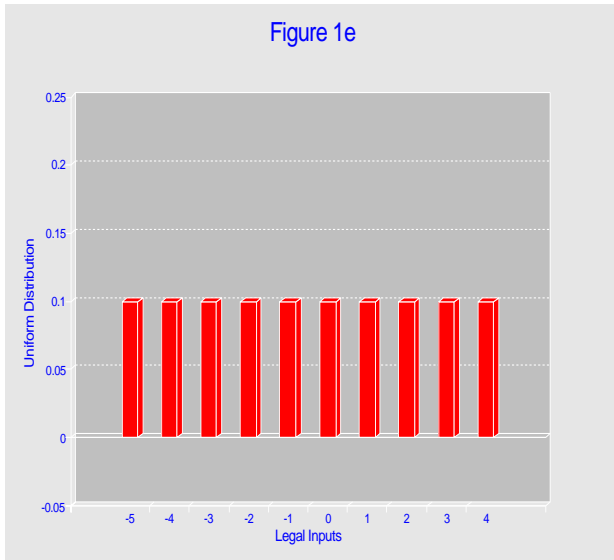
these bins are, therefore, more likely to contain undiscovered faults. However, if the software has high fault-tolerance when executing low probability input test cases, faults are likely to remain hidden during operation; i.e., the software will tolerate problems that arise during the operational life-time of the code. It is this argument that has motivated our work in this area.

In traditional reliability testing with an input distribution, repeated executions (hopefully) mimic the conditions that the software will experience in operation. Reliability information can then be predicted under assumptions about the accuracy of the input distribution. Testing with  $\bar{Q}$  does not give this “predicted” reliability. Instead, testing with  $\bar{Q}$  will exercise the code with less frequently (but still legal) selected inputs. As such, the tester can expect to discover soft-

ware faults that might linger, undetected, for quite some time when the software is tested or used by customers under situations anticipated by the developers. (For example, an incorrect exception handler might be caught if an unlikely input that exercises the handler is chosen.) However, these undetected errors can surface either after time works against the probabilities or if the distributions shift during use.

$\bar{Q}$  could be particularly helpful when software has passed all planned tests using  $Q$ . Faults that are only sensitive to infrequent inputs (according to  $Q$ ) could be more likely to be uncovered by  $\bar{Q}$ , particularly if the reason such faults are “good” hidiers is that they are rarely executed according to  $Q$ , but they are reached by  $\bar{Q}$ .

Another valuable application of EPA based on  $\bar{Q}$  is



to predict a likely probability of failure for a piece of software. The smallest remaining errors are the most difficult to find, and traditional testing is limited in giving assurance about very “small” faults (i.e., those that result in a very small probability of failure). EPA can help in this regard: if EPA can demonstrate that very large faults are unlikely (i.e., faults will tend to be “small,” and therefore unlikely to be triggered during operation), then an additional confidence can be added to the original squeeze play confidence (based on  $Q$  and testing) that no failures will occur [3, 12].<sup>7</sup>

<sup>7</sup>EPA with  $\bar{Q}$  strengthens the squeeze play; even if a location is likely to reveal faults with  $Q$  and does not, and it is fault-tolerant to faults with  $\bar{Q}$ , then we have two pieces of information suggesting that during the life of the code according to  $Q$ , failures will not occur. Whenever, according to the op-

## 2 Conclusions

Information concerning how problems propagate through safety-critical systems and the probabilities of propagation occurring is useful for fault-tolerance assessment. For fault-tolerance, regions of high propagation are dangerous, because they are likely to produce erroneous values that will lead to unsafe conditions.

This paper presents the theory behind a fault-tolerance assessment method for dynamically studying the effects of simulated hardware failures and software faults when less likely operational inputs are selected.

small sized faults can be shown to be unlikely to exist and the program appears to be fault-tolerant in the ultra-reliable region of the input space, i.e., the improbable input subdomain, the squeeze play becomes more practical.

This methodology is empirical, not formal, and thus the results from such a method are not absolute guarantees of how the system will behave when deployed, but rather predictions that are based on prior observations. But although empirical, this methodology is not testing, and thus no oracle is required.

We should also mention that the ideas that we have presented here can be immediately applied to issues associated with *software security*. Many of the same development methodologies used for software safety are applied to computer security. An example is the building of *firewalls* in a security sensitive application to protect against malicious threats. The method outlined here could easily be customized for computer/software security applications.

### 3 Future Work

EPA is currently automated as one tool in the *PiSCES Software Analysis Toolkit* (TM) 1.5 [9]. The EPA tool queries the user to specify which class of faults they want to simulate: incoming hardware failures, software faults, or a combination thereof. The EPA tool allows the user to define  $\mathcal{C}_P$ . If members of  $\mathcal{C}_P$  are ever observed, the user knows where in the code such a problem was able to propagate from, and the user also knows how frequently this event was observed. The tool also allows the user to estimate a mean-time-to-hazard (MTTH) for each class of catastrophic events. We are currently incorporating the distribution inversion capability into the tool for several large commercial applications that are scheduled to be analyzed in 1995. The main effort here is in modifying our algorithm to handle multi-dimensional input distributions.

### Acknowledgement

The authors wish to thank the referees for their valuable comments on an earlier draft. This research was partially funded by NASA Contract NAS1-20388, National Institute of Standards and Technology Contract 50-DKNA-4-00119, National Science Foundation Grant DMI-9461928, and NASA Grant NAG1-884.

### References

[1] N.G. LEVESON AND P.R. HARVEY. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[2] D. J. LAWSON. Failure Mode, Effect, and Criticality Analysis. In J. K. Skwirzynski, editor, *Electronic Systems Effectiveness and Life Cycle Costing*, pages 55–74, NATO ASI Series, F3, Springer-Verlag, Heidelberg, 1983.

[3] J. VOAS AND K. MILLER. Improving the Software Development Process Using Testability Research. In *Proc. of the 3rd Int'l. Symposium on Software Reliability Engineering.*, pages 114–121, Research Triangle Park, NC, October 1992. IEEE Computer Society.

[4] J. VOAS AND K. MILLER. Dynamic Testability Analysis for Assessing Fault Tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.

[5] J. D. MUSA. Operational Profiles in Software Reliability Engineering. *IEEE Software*, 10(2), March 1993.

[6] NASA. NASA Software Safety Standard. Office of Safety and Mission Assurance, June 1994. Interim Report 1740.13.

[7] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987.

[8] R. S. PRESSMAN. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Book Company (New York), third edition, 1992.

[9] Reliable Software Technologies Corporation. *PiSCES Software Analysis Toolkit (TM) User's Manual*, December 1994. Version 1.5.

[10] UNDERWRITERS LABORATORY INC. Safety Related Software, January 1994. Standard for Safety UL1998, First Edition.

[11] J. VOAS. *PIE: A Dynamic Failure-Based Technique*. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

[12] M. FRIEDMAN AND J. VOAS. *Software Assessment: Reliability, Safety, and Testability*. to be published by John Wiley and Sons, New York, 1995.