

Predicting Software’s Minimum-time-to-hazard and Mean-time-to-hazard for Rare Input Events

Jeffrey M. Voas
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
(703) 404-9293
jmvoas@RSTcorp.com

Keith W. Miller
Department of Computer Science
Sangamon State University
Springfield, IL 62794
(217) 786-7327
miller@eagle.sangamon.edu

Abstract

This paper turns the concept of input distributions on its head to exploit *inverse* input distributions. Although such distributions are not always true mathematical inverses, they do capture an intuitive property: inputs that have high frequencies in the original distribution will have low frequencies in the inverse distribution, and vice versa. We can use the inverse distribution in several different quality checks during development. Here, we will provide a fault-based (fault-injection) method to determine minimum-time-to-failure and mean-time-to-failure for software systems under normal operational and non-normal operational conditions (meaning rare but legal events). In our calculations, we will consider how various programmer faults, design errors, and incoming hardware failures are expected to impact the observability of the software system.

1 Background

Software engineers who are concerned with software reliability estimation traditionally use information about the likely distribution of inputs during use. Such a distribution is termed an *operational profile* [9, 7]. 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. This paper provides a method for quantifying the fault-tolerance of a system when it encounters the rarest operational events. Current testing/reliability methodologies attempt to show that a

program is safe under *frequent* operational conditions; we demonstrate fault-tolerance under *infrequent* operational conditions without performing testing, which is a much more difficult software quality characteristic to demonstrate.

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 a distribution is not necessarily a true mathematical inverse, it captures an important intuitive property: elements that occur frequently in the original operational distribution, Q , occur infrequently in the inverse operational distribution, \bar{Q} ; elements that occur infrequently in Q occur frequently in \bar{Q} . In this paper, we also provide formulae that calculate minimum-time-to-hazard and mean-time-to-hazard when Q and \bar{Q} are used to assess fault-tolerance.

Our meaning of the term *software fault tolerance* is different from the traditional hardware definition.¹ This paper considers fault-tolerance to be a measure of a program’s ability to:

1. compute the correct output even when the program itself suffers from incorrect logic,
2. compute the same output when the program itself receives *corrupted* incoming data during execution as it would if the data were not corrupted, *and*

¹The 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.

3. produce non-hazardous outputs, regardless of the circumstances.

Widely accepted software engineering design practices have argued for *robustness* and *graceful degradation* when a system gets into an undesirable state [10]. 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, but still 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 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-effect of designing for robustness.

2 Extended Propagation Analysis for Fault-Tolerance Measurement

Our fault-tolerance assessment method uses inverted operational distributions and relies heavily on a technique termed “extended propagation analysis” (EPA) [6].² 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. 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.

²EPA is a spin-off of Voas’s Sensitivity Analysis technique [13]; the main differences are that EPA is only concerned with the propagation condition, and EPA is also concerned with incoming hardware failures. Sensitivity Analysis is only concerned with resident program faults. Also, EPA differentiates classes of failure; Sensitivity Analysis does not.

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 software output if a hardware sensor that feeds data to the software were to malfunction or if the software itself were to malfunction (with respect to the specification). Based on the fault classes we simulate, we assess the net impact of faults on critical output variables and predict whether the impact is likely to produce a catastrophe.

2.1 The EPA Theoretical Model

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 , Q 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 . And let \mathcal{A}_{lPx} represent the data state produced after executing location l on the i^{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:

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

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^{th} iteration of location l on input x ; $\check{\mathcal{A}}_{lP_i x}$ 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 can lead to a catastrophe. When a system gets into a bad (undesirable) state during execution, the next event that we would like to occur is recovery from that state back to an acceptable state. Otherwise we would like to demonstrate that the bad state has no damaging consequences. Simulated infections are the mechanisms that are employed in EPA to

allow observation of the impact of different classes of undesirable states. Simulated infections mimic the effect of both programmer faults and hardware failures that result in spurious inputs into a system. Typical classes of hardware failure-based simulated infections include: (1) all bits high, (2) all bits low, (3) random offsets, (4) completely random bits, and (5) time modifications (speed up and slow down). Typical classes of programmer faults that are used in this fault-injection based method are discussed in [13].

Simulated infections are implemented 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 a data state as an incoming parameter, changes it according to certain parameters that are either input to the function or compiled constants, 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*.

A program 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 $\alpha_{lP\Delta}$. ([13] explains how to handle the possibly non-recursive and infinite nature of $\alpha_{lP\Delta}$). 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 $\Pi_{alPQk} = \mathbf{variable_set}$. (Π_{alPQk} represents the set of output variables that have different values given execution of P occurs with $\check{\mathcal{A}}_{lPx}$. Π_{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, where n is left up to the user.

Step 5 of Algorithm 1 and similarly Step 3 of Algorithm 2 are very critical to the success of 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,” [12] defines the following classes that must be simulated in Step 5 to demonstrate compliance with 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 the risks associated with the classes 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 3). 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 [8]. These errors 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.)

There will be instances, however, where we are not necessarily concerned 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 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 $\alpha_{lP\Delta}$. ([13] explains how to handle the possibly non-recursive and infinite nature of $\alpha_{lP\Delta}$). 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*).

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

We now take the information provided by EPA 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 specific catastrophic failures could originate.

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 internal set 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 the EPA algorithm 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 .³ 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 the EPA 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 ls , 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.

³ \mathcal{C}_P is determined during hazard analysis in the requirements phase and is directly related to system safety requirements.

2.3 Examples

Our first example of how EPA operates will be demonstrated by the following scenario. Suppose that our goal is to answer the following question: “if a plane is in landing mode and the altimeter reports to the flight-control system that the plane is 50 feet from touchdown when the plane is really 150 feet from touchdown, will there be any impact on either the control of the engines or control of the wing flaps?” Here we are concerned with the class of hardware failures associated with a broken altimeter, and we are concerned with the output commands from the flight control system to the flaps and the engine.

The algorithms for EPA allow us to simulate a broken altimeter by modifying the altimeter output to the flight-control software. After the algorithm is told that we are concerned with the output variables that control engine and wing-flap commands, we can then be more specific and tell EPA that we are only concerned if these variables are affected in a specific manner that could put the plane into an unsafe landing state. If this is done, EPA can report not only whether the engines or flap commands will be affected, but the frequency with which such occurred. Hence we can empirically demonstrate if a broken altimeter is a software safety threat to a plane in that landing situation.

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

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

As you can see, care is required when we define a catastrophic event. The less ambiguous the description, the more precise the result. 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:

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 its 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 $PRED$.

Unperturbed, No Violation Nothing to report here.

Perturbed, $PRED$ Violated This will be considered as a catastrophic failure; observing this event does not require executing the unperturbed version.

Perturbed, Corrupted, User did not specify $PRED$

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 to:

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

This section has described how EPA can be used to simulate software and hardware faults. EPA requires humans to make two determinations: safety descriptions in terms of the variables, and the input distribution. After these determinations are encoded, EPA works without human intervention. In the next section we discuss how an input distribution can be transformed into a second distribution: an “inverted” distribution. Later, we’ll see how EPA and inverted distributions can be used together.

3 Inverted Distributions

Quantitative assessment of software quality is often based on experiments, such as testing. But these experiments require some knowledge of how the software will be used. Specifically, we need knowledge of an “input distribution,” (preferably an *operational profile* [7]). Using the input distribution is essential to some goals, but it has inherent weaknesses for others.

For example, if a particular input value has a very small but non-zero probability of selection, and if that input is the only input that will activate a fault in the code, then random testing is unlikely to uncover that fault. In this paper we explore a technique that uses the input distribution in a novel way to complement traditional uses of the input distribution: we invert the input distribution.

Assume that during testing with inputs from Q , no failures are observed and the number of test cases used are relatively few (with respect to the possible input values).⁴ These successful tests allow us to estimate relatively modest levels of reliability. Test cases selected according to \bar{Q} that compute correct results would allow for an even higher reliability estimate. To be able to gather this additional information, we have three means available:

1. Test according to \bar{Q} .
2. Perform EPA with \bar{Q} , showing that the selected members are incapable of causing the output state to be affected by both internal design flaws and corrupted input values that are coming into the system from external sources.
3. A combination of 1 and 2.

In the remainder of this paper, we will focus on the implications of performing the second alternative, and what possible impact quantifying the fault-tolerance of \bar{Q} might have on predictions of minimum-time-to-hazard and mean-time-to-hazard. Realize that during the operational life-time of the software, those members likely to be selected according to \bar{Q} will almost certainly be fed into the system as inputs at some point.

Our intuition for the value of inverted distributions follows: \bar{Q} provides quick access to those test cases that would only be selected if intractable numbers of test cases were selected according to Q . When we select 1,000 test cases, 990 of those test cases come from the regions of the input space that are the *most* likely, and 10 test cases come from the regions that are somewhat less likely. It is almost impossible that in our sample of 1,000, we have any test, z , whose likelihood to be selected according to Q is 10^{-6} . When we select another 9,000 test cases for a total of 10,000 test cases, 9,900 of those test cases come from the regions of the input space that are the most likely. Essentially all we

did was to select 8,910 more test cases from the highest probability regions of Q , and we probably still have not selected z . So what do we truly learn about our software’s quality when we increase our test suite size by another order of magnitude? We “hopefully” learn that the code still works correctly for the regions of Q that it already worked correctly for when we sampled earlier, likely tests. But for extremely low probability inputs, even 10,000 samples may not be enough to likely select them. Inverted distributions increase the likelihood of sampling those rare test cases without waiting to sample those test cases along with the likely test cases.

3.1 Inverting One-dimensional Input Distributions

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 [7], 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.

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 $Q(i) = 0$ really means $Q(i) \approx 0$ because i 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 properties of a probability density function make this difficult to attain. Instead, we offer the following constructive definition of the “inverse,” \bar{Q} :

⁴These are the inputs with the greatest likelihood of selection in 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 $g'(i) = 2 * M - Q(i)$. This reflects the original distribution about the mean.
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 $g''(i) = (g'(i) + \mathbf{abs}(m))/(1 + N * (\mathbf{abs}(\mathbf{m})))$. This step translates the distribution so that each point is non-negative, then normalizes the function to have area 1. Then g'' is \bar{Q} .

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

In the algorithm, the first step indicates the average probability of selection for each legal input. This average probability line 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 this distribution, we reflect the distribution about the uniform distribution line

For some Q s, the inverse of \bar{Q} will return Q . However, not all input distributions are so nicely behaved. Some Q s, when reflected about the uniform distribution, result in negative probabilities, which are undefined. 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 graphs up by a constant equal to the absolute value of the smallest negative value in the y direction. This graph then is non-negative, but the y values no longer add to one, so our second step requires normalization, i.e., dividing each y value by the sum of all the y values.

For these more complex situations, the inversion of the inversion will not yield the original distribution. Also, the normalizing step makes the resulting distributions to appear “flatter” than we would like, losing some of the shape characteristic of the original distribution. This flattening has an explanation

tied to testing. If the original testing emphasized a small subset of the legal inputs, then an inverted distribution cannot give this same attention to the rest of the inputs; instead, this attention must be spread out among a larger group of inputs, thus the flattening of the distribution. There are alternative methods of solving the negative probability problem, but the scheme described above is practical and, we contend, useful. Note that any inputs that share the largest probability in the original distribution will have zero probability in the inversion.

3.2 Inverting Multi-dimensional Input Distributions

Inputs to a program are, in general, *multi-dimensional*. If the input to a program is one or two variables, the input distribution can be viewed conveniently as a graph.

The inversion algorithm described above does generalize for an input space of n dimensions:

Algorithm 4:

1. Describe the probability of each input (an ordered n -tuple) as a value in the $n + 1$ dimension.
2. Determine the hyper-plane which is defined by setting the $n + 1$ dimension value to a constant, $1/K$, where K is the cardinality of the input space. (That is, the average probability of selection in a uniform distribution of all legal inputs.)
3. Reflect the input distribution about this hyperplane.
4. If any of the resulting values in the $n + 1$ dimension are negative, add a constant C to each value, translating the graph so that all the values in the $n + 1$ dimension are non-negative.
5. Normalize the resulting graph in $n + 1$ space, dividing each value by the total volume. Now the value in $n+1$ space associated with each n -tuple is the probability of selection in \bar{Q} .

3.3 \bar{Q} and EPA

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 it is controlling. If either type of event 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 or

$$(\exists k \mid 1 \leq k \leq n)[(\exists \gamma \mid \gamma \in \mathcal{C}_P) (\gamma \subseteq \Pi_{alP\bar{Q}k})] \vee PRED \quad (4)$$

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, depending on which definition of catastrophic that we use. Our innovation will include the utilities necessary to perform fault-tolerance assessment on single and multi-dimensional input distributions.

3.4 \bar{Q} and Traditional Testing

Our focus to this point has been on using \bar{Q} to assess fault-tolerance. Here, we will briefly step back and analyze what might be gained if system level testing is performed according to \bar{Q} . 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 input distribution. Testing with \bar{Q} does not predict reliability. Instead, it would exercise the code with less frequently (but still legal) selected inputs.⁵ As such, the tester can expect to discover software 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.) These undetected errors can surface either after time works against the probabilities or if the distributions shift during use.

4 Mean-time-to-hazard and Minimum-time-to-hazard

We now wish to use the output information from Algorithms 1 and 2, Π_{alPQk} and $\hat{\psi}_{alPQ}$, to pre-

⁵Testing with \bar{Q} is not equivalent to *stress testing* as defined in Beizer [3], which tests a system according to abnormally high numbers of processing requests in a short time interval. But it does have the same intuitive appeal.

dict software quality. Here, we will provide quantitative risk metrics that predict *mean-time-to-hazard* (MTTH) and *minimum-time-to-hazard* (MinTTH). Our predictions will be based on program execution frequency per unit time, the test distribution, and the faults that are injected via EPA. Of these three key parameters, the parameter that is the most likely to be biased is the parameter that is based on fault-injection. Hence the accuracy of the predictions (in terms of actual clock time before a true hazard might occur) rests heavily on how closely the injected faults behave like real-life anomalies. For a particular system, this will not be known early in the system's life-time, but it may be measurable later in the system's life after anomalies are detected and debugged. Regardless, this metric does provide a way of comparing different systems, and auditing the fault-tolerance of successive versions of a system.

4.1 Mean-time-to-hazard and Minimum-time-to-hazard for Q and \mathcal{C}_P

We begin by first assuming that $PRED$ has not been specified, and then later in Section 4.2 we generalize it for $PRED$. *Mean-time-to-hazard* is defined as the average time interval before a catastrophic event will occur based on Q , the classes of fault injections that EPA used, and the classes of catastrophic events that EPA considered. MTTH is simply the average amount of time between our observing catastrophes, given Q , the injected fault classes, and the defined catastrophic events. The *minimum-time-to-hazard* is the shortest predicted period of time before any catastrophic event defined in \mathcal{C}_P will occur, i.e., this is the catastrophic event that would be expected to occur the soonest given Q , the injected fault classes, and the defined catastrophic events in \mathcal{C}_P . Our equations for $MTTH_Q$ and $MinTTH_Q$, given normal operational profiles, follow:

$$\theta_Q = \frac{\sum_{i=1}^n f([\exists \gamma \mid \gamma \in \mathcal{C}_P) (\gamma \subseteq \Pi_{alPQi})]}{n} \quad (5)$$

$$f(x) = \begin{cases} 0 & \text{otherwise} \\ 1 & \text{if } x \text{ is true} \end{cases}$$

$$MTTH_Q = [\theta_Q \cdot (\frac{\text{number of program executions}}{\text{unit of time}})]^{-1} \quad (6)$$

5 Summary

Traditionally, the input distributions used during testing engender the most used functions to be the most well tested (and thus the least likely to fail) functions. But for life-critical applications, for software that has already been subjected to intense testing using the original input distribution, or for software that has already been in the field for some time, a tester may be more interested in faults that occur infrequently. For example, the recent Pentium problems surfaced because relatively infrequent situations (at least according to Intel) tripped a design flaw. One billion tests had failed to reveal the flaw [1]. INTEL had done both mathematical modeling and experiments to confirm their failure rate predictions, however the number of test cases needed to validate the analysis was over one trillion test cases [2]. Testing with an “inverted” distribution might have caught those problems before release.

We have written elsewhere of the importance of hidden faults when trying to quantify the reliability of software [5, 15, 4]. Faults that are unlikely to cause failures during testing are particularly problematic when we want to verify extremely low probabilities of failure (the ultra-dependability problem) [14]. One of the reasons that faults may hide from testing is that the affected code is infrequently executed during testing. By using inverted distributions, we can explore this possibility experimentally. By concentrating on inputs that are infrequently executed by the original input distribution, we can adjust our estimates of the probabilities that catastrophic failure can occur.

Information concerning how problems propagate through safety-critical systems and about how the probabilities of propagation occur 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 new fault-tolerance assessment method for dynamically studying the effects of simulated hardware failures and software faults when less likely operational inputs are selected. 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. This scheme should be more beneficial when the original distribution has noticeable spikes, i.e., far from uniform, since the inverse of a uniform distribution is the uniform distribution.

$$\text{MinTTH}_Q = [\phi_Q \cdot (\frac{\text{number of program executions}}{\text{unit of time}})]^{-1} \quad (7)$$

$$\phi_Q = F(\mathcal{C}_P, \bigcup_{i=1}^n \Pi_{alPQi})$$

$$F(\alpha, B) = \begin{cases} g(\gamma) & \text{if } ((\exists \gamma \mid \alpha \in B)[(\forall \rho \mid (\rho \in B) \wedge \\ & (\rho \neq \gamma)) (g(\gamma, B) > g(\rho, B))]) \\ & \text{is true} \\ 0 & \text{otherwise} \end{cases}$$

$$g(\rho, \mathcal{Z}) = \begin{cases} \left| \bigcup_{j=1}^n h(\rho, j) \right| & \text{otherwise} \\ 0 & \text{if } \mathcal{Z} = 0 \text{ is true} \end{cases}$$

$$h(\sigma, \nu) = \begin{cases} \sigma & \text{if } \sigma \subseteq \Pi_{alPQ\nu} \text{ is true} \\ \emptyset & \text{otherwise} \end{cases}$$

4.2 Mean-time-to-hazard and Minimum-time-to-hazard for Q and $PRED$

We now provide the mean-time-to-hazard calculations assuming that the user has decided to use Algorithm 2 that uses $PRED$ (instead of Algorithm 1 that builds Π_{alPQk}).

$$\text{MTTH}_Q = \left[\left[\frac{1}{M} \sum_{l=1}^M \hat{\psi}_{alPQ} \right] \cdot \left(\frac{\text{number of program executions}}{\text{unit of time}} \right) \right]^{-1} \quad (8)$$

$$\text{MinTTH}_Q = \left[\max_l [\hat{\psi}_{alPQ}] \cdot \left(\frac{\text{number of program executions}}{\text{unit of time}} \right) \right]^{-1} \quad (9)$$

4.3 Generalize to \bar{Q}

The formulae provided in Sections 4.1 and 4.2 can be based on \bar{Q} instead of Q if \bar{Q} is used during EPA.

Although empirical, this methodology is not testing, and thus no oracle is required.

6 Future Work

EPA is currently automated as one tool in the *PiSCES Software Analysis Toolkit*^(R) [11]. We are currently incorporating the distribution inversion capability into the tool for several large commercial applications that are scheduled to be analyzed in 1996. The main effort here is in modifying our algorithm to handle multi-dimensional input distributions. To date, we have only looked at applying these ideas to numerical input data, and future work will look at applying these notions to more complex and varied test data. On the theoretical side, we are convinced that fault-tolerance measurements with \bar{Q} have a role in improving the confidence in software and thus should be included in the “Squeeze Play” dependability model [5, 15], but we have not yet explored how to do so.

Acknowledgment

The authors are grateful for the careful and insightful comments made by the reviewers that greatly improved this paper. The authors also thank Aaron Binns for his help with the formulae. This research was partially funded by NASA Contract NAS1-20388, NASA Grant NAG-1-884, NIST Contract 50-DKNA-00119, and U.S. Air Force Contract F30602-95-C-0158.

References

- [1] H. P. SHARANGPANI AND M. L. BARTON. Statistical Analysis of Floating Point Flaw in the Pentium Processor. INTEL Corporation white paper, November 30, 1994.
- [2] B. BEIZER. Software Engineering is Illegal. Address at Quality Week. June 2, 1995. San Francisco, CA. .
- [3] B. BEIZER. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd. edition, 1990.
- [4] J. VOAS, C. MICHAEL, AND K. MILLER. Confidently Assessing a Zero Probability of Software Failure. *High Integrity Systems Journal*, 1995. To appear.
- [5] 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.
- [6] J. VOAS AND K. MILLER. Dynamic Testability Analysis for Assessing Fault Tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.
- [7] J. D. MUSA. Operational Profiles in Software Reliability Engineering. *IEEE Software*, 10(2), March 1993.
- [8] NASA. NASA Software Safety Standard. Office of Safety and Mission Assurance, June 1994. Interim Report 1740.13.
- [9] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987.
- [10] R. S. PRESSMAN. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Book Company (New York), third edition, 1992.
- [11] Reliable Software Technologies Corporation. *PiSCES Software Analysis Toolkit (TM) User's Manual*, December 1994. Version 1.5.
- [12] UNDERWRITERS LABORATORY INC. Safety Related Software, January 1994. Standard for Safety UL1998, First Edition.
- [13] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.
- [14] K. MILLER, L. MORELL, R. NOONAN, S. PARK, D. NICOL, B. MURRILL, AND J. VOAS. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.
- [15] R. HAMLET AND J. VOAS. Faults on Its Sleeve: Amplifying Software Reliability Assessment. In *Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis '93*, pages 89–98, Cambridge, MA, June 1993.