Fault Detection Effectiveness of Spathic Test Data

Jane Huffman Hayes
Computer Science Department
Lab for Advanced Networking
University of Kentucky
301 Rose Street
Lexington, KY 40506-0495 USA
+1 859 257 3171
hayes@cs.uky.edu

Pifu Zhang
Computer Engineering Department
Changsha Communications
University
42 Ciling Road
Changsha, Hunan, 410076 China
+86 731 521 9108
pifuzhang8@hotmail.com

Abstract

This paper presents an approach for generating test data for unit-level, and possibly integration-level, testing based on sampling over intervals of the input probability distribution, i.e., one that has been divided or layered according to criteria. Our approach is termed "spathic" as it selects random values felt to be most likely or least likely to occur from a segmented input probability distribution. Also, it allows the layers to be further segmented if additional test data is required later in the test cycle.

The spathic approach finds a middle ground between the more difficult to achieve adequacy criteria and random test data generation, and requires less effort on the part of the tester. It can be viewed as guided random testing, with the tester specifying some information about expected input. The spathic test data generation approach can be used to augment "intelligent" manual unit-level testing. An initial case study suggests that spathic test sets detect more faults than random test data sets, and achieve higher levels of statement and branch coverage.

1. Introduction

Test data is a common denominator of all testing. Test data may be generated in order to maximize a particular test adequacy criterion, it may be randomly generated, it may be manually developed by a tester who possesses vast amounts of experience with the application under test. Regardless of testing at the unit or system level, test data that exposes program faults is required. Testing poses a question, is the test data "good" enough? Efforts aimed at developing an adequate test set may usurp valuable test execution time.

The allocated time period for testing often includes test set development and the testing itself. In order to stay within the allotted time period or to develop more code, unit testing may not be performed or may be assumed to be included in integration testing. Within the framework of testing, negative testing, i.e., testing with rarely expected test values, and positive testing, i.e., functionality testing, should be employed. It is also desirable to know how many test cases should be executed. If we are able to specify the number of test cases required in advance of the testing execution, we can better estimate the test period.

We describe a new technique for unit testing. The approach is called "spathic," meaning layered or stratified, because a probability distribution is divided into layers and test values are selected from these layers. The technique incorporates reliability-oriented methods and "intelligent" testing [7] and is intended to provide higher levels of defect detection and comparable levels of structural coverage as random testing. The technique involves examining an input distribution related to a given unit under test and segmenting or stratifying it based on the goal of testing. For example, if negative testing is being performed, the input distribution is segmented to focus on the least likely inputs. Test data are then selected from the layers.

We report an initial case study in which test data was generated for numerous programs as well as their fault-seeded versions. Spathic test data generation produced test data that resulted in similar levels of structural coverage and higher fault detection abilities than did simple random sampling, without requiring larger test sets.

The paper is organized into six sections. Reliability testing and unit testing are discussed in Sections 2 and 3, respectively. Our technique for generating test data is presented in Section 4. Section 5 contains results of the case study. Finally, conclusions and future work are presented in Section 6.

2. Reliability Testing

Some researchers have categorized testing techniques into synthetic and operational [12]. Synthetic testing involves systematic selection of test data whereas operational testing has the intended users employ the

program in the field as they see fit. Synthetic data is generated from non-operational means. Operational test data refers to data that has been generated using specific knowledge of the software, i.e., an actual usage profile [23]. Software reliability engineered testing [15,16] is an example of operational testing.

Software reliability engineered testing uses an operational profile or usage model to guide its executions and then measures the failure times to estimate the reliability of the system in the field. A good operational profile or operational input distribution is key to the process. The profile is a characterization of the program's expected use. As the profile is a probability distribution there is a probability of selection associated with each point, with all the probabilities summing to one. Software reliability engineered testing then goes further to examine the failure density for a program. The failure density, or failure probability, is the probability that the program will fail on a random input. This technique estimates the reliability of a software system and is utilized after a system has been developed and integrated. Note that this technique has been found unsuitable for life-critical real-time applications [22].

The emphasis of this testing is estimating the reliability of a program in the field. The better the operational profile, the better the reliability estimate. Developing a good operational profile is very difficult. For many programs, it would be cost- and time-prohibitive to develop an operational profile that provides point-by-point probabilities. Also, a large number of tests are needed to achieve a modest level of confidence at a modest failure density bound [5,6,7]. To reduce the number of tests needed, it would be desirable to partition the system's input domain into equivalence classes, such that a test of a representative of each class can be viewed as testing the entire class. But we know that in general this is unattainable [19]. Some approaches have been offered to assist with these problems. Woit presents a general method of operational profile specification allowing conditional and unconditional input dependencies to be specified [20]. Podgurski et al [12] use stratified sampling to produce more accurate estimates of failure frequency by collecting execution profiles of beta-test executions and applying cluster analysis to the profiles to partition the executions. They selected a stratified random sample of the executions and used them to estimate the proportion of failure in the entire population of captured executions.

Ntafos [9] builds on Howden's findings [19] that we want to produce homogeneous partitions or subdomains, those in which all inputs either result in a failure or produce correct results [9]. If we can achieve some level of homogeneity, we can take advantage of the well-known statistics result that stratified random sampling can lead to precise estimates using a smaller sample than simple random sampling [2,9]. Ntafos also presents some of the shortcomings of partition testing: to achieve a true proportional allocation may be very costly because of the large number of test cases needed; and with a large number of test cases, random sampling allocation and proportional partition allocation tend to become similar [9]. Even if the number is large, it is often useful to know how many test cases will be required to satisfy a given objective. Statistical testing techniques have also been used to estimate the number of test cases needed to achieve a particular test objective (such as coverage) [17,18].

With this in mind, we look at testing in the unit case. From practical experience, we know that testers will generally not run large numbers of test cases when unit testing. Our approach seeks to generate a reasonable number of test cases to overcome the limitations pointed out by Ntafos and others as well as to ensure that our technique is practical. We use some aspects of software reliability engineered testing, but at the unit level. We use an input distribution and sample over that distribution. We use a form of stratified sampling, but we stratify or partition the input space, not the execution profiles. We look at disjoint subdomains, but they have not been selected based on significant analysis such as in "intelligent" testing [7]. As compared to reliability testing, we are not concerned with failure density, but with fault detection effectiveness. Our approach uses an input distribution at the unit or component level and allows the tester to guide the selection of random inputs without requiring knowledge of a special notation or formal specification technique. Having discussed operational testing, we now turn to synthetic testing. But first, we will establish unit testing as our context.

3. Approaches to Unit Testing

In this paper we consider a unit to be the smallest compilable element of an application[1]. Unit testing is often overlooked or replaced with integration testing. These are not advisable practices since unit testing provides the opportunity to perform thorough structural testing, concentrating on the internal structure and implementation details of the small element.

Recalling our dichotomy of techniques as operational or synthetic, Podgurski et al propose that synthetic testing does not provide a good estimate of reliability, but may help to make software reliable enough to proceed to operational testing [12]. Also, we saw in the previous

[1] Note that this may be a class in Java and that some view such testing as integration level testing. In this paper, we consider it to be unit testing.

section that operational testing occurs at the system level. Our concentration is on unit testing, so we will look to synthetic testing for assistance. Though many types of synthetic testing have been developed and examined in the literature, it is important to find better ways to generate synthetic test data. We now examine several synthetic techniques that can be applied to unit testing.

3.1 Structural Testing

Many structural testing techniques exist such as statement testing, branch testing, def-use association testing [14]. These techniques generate test cases in order to maximize compliance with a test adequacy criterion. For example, branch testing generates test cases such that every branch in a unit is exercised at least once. If the tests do execute every branch they are said to achieve 100% branch coverage.

3.2 Specification-based Testing

Specification-based testing is a method that concentrates on what software is supposed to do as opposed to how it is accomplished. This is in contrast to structural testing that focuses on specific structural components of a program, generally in keeping with a selected structural testing criterion, and ensures that they are adequately exercised. Specification-based testing is often referred to as "black box testing" and structural testing is referred to as "white box" or "glass box" testing.

Equivalence testing decreases the number of tests one must execute through the use of equivalence classes or equivalence partitions. An equivalence class is a subset of a program's input or output domain such that testing of a single item of the class theoretically tests that entire class. Testers examine both the program structure and specification in order to develop equivalence classes.

A related specification-based testing method that we have seen used in industry is category-partition testing [1,11]. In category-partition testing, the engineer defines categories for the program under test and then partitions these categories into equivalence classes. An input is then selected from each equivalence class, called a choice [1,11]. Ammann and Offutt [1] extended this by defining a coverage criterion for category-partition test specifications using formal schema-based functional specifications.

Programs that have been represented at some point in the lifecycle by formal specification are excellent candidates for specification-based testing. For example, the formal specification may be in an executable form, such as a finite state machine or a state chart. Aspects of such formal models can be represented in the form of a directed graph. This allows us to select test values to "cover" the graph. That is, the program-based adequacy criteria based on the flow-graph model can be adapted for specification-based testing [Fujiwara et al. 1991; Hall and Hierons 1991; Ural and Yang 1988; 1993].

3.3 Positive/NegativeTesting

We have noted that many industrial development undertakings characterize testing as either positive (sometimes called constructive) or negative (sometimes called destructive). Positive testing has as its goal to demonstrate that software performs its intended functions, though we can never prove correctness [19]. Test values that are most likely to be encountered operationally are used. Developers often perform this testing on their own software. Negative testing attempts to "break" the software and to uncover weaknesses in the application. Here, test values that are least likely to be encountered are often used. Negative testing may be performed by an independent organization (not the developers). Positive and negative testing can be applied at any level of testing, unit through system.

Though a good testing process combines both of these types of testing, a conceptual separation of the two may result in better test data and therefore better testing. In requirements engineering, separation of what/how has helped us to focus on the problem to be solved while delaying implementation details. Also, the aggregation/decomposition perspective has helped us to reduce the complexity of an overall system by breaking it into smaller elements that we can understand all at once. Similarly, for testing we can view the generation of negative/positive test data as a viewpoint or perspective. This viewpoint may cause us to think of other test cases that we may not have otherwise considered.

3.4 Tester Effort

It is clear that the aforementioned testing techniques fall under the category of "intelligent" or guided testing, i.e. testers make choices of tests that are more likely to reveal faults than random sampling over an operational distribution [7]. However, depending on the testing method selected, a great deal of time and effort must be exerted by a tester to develop the required test cases. There are testing tools available to help automate aspects of test data generation, but the task can still be daunting and time consuming. It has been our experience that it is relatively easy to obtain 80% coverage for numerous criteria (such as branch), but that to generate the additional data required to achieve 90-100% coverage is difficult. Also, it can be the

.

case that large numbers of test cases are required to achieve high levels of adequacy criteria.

As discussed in section 2, our approach seeks to generate a practical number of test cases, such that a tester would execute them all even without tool support. We use aspects of equivalence- and category-partition testing in that we have the tester characterize the input space, the input space is then segmented or layered, and a criterion is used to select points from within the layers. We have evaluated the structural adequacy of our generated test data and found it to achieve similar levels of branch and statement coverage as random sampling.

4. Spathic Test Data Generation

The spathic test data generation approach is an "intelligent" random testing method. It is suitable for unit testing and may also be suitable for integration testing, though not explored in this paper. The spathic method provides testers a procedure for generating test data. The tester has two main tasks: 1) to indicate the type of test data desired, and 2) to characterize the input space. The approach is detailed below.

4.1 Process

The steps in the spathic test data generation method that result in test data are as follows.

1. Determine the type of data desired as either most probable (positive) or least probable (negative).
2. Briefly analyze the specification to understand at a high level what the component does.
3. Characterize the input domain by:
    a. selecting an input probability distribution (Gaussian, Poisson, etc.)
    b. specifying the number of test values desired, and
    c. specifying the mean and deviation from the mean.
4. Solve several equations to generate the test data (or use our tool to solve the equations and generate the data).

We now examine these steps at a high level and then present the underlying mathematics. Let us examine steps 2 through 4 and then come back to step 1. In step 2, the tester needs a high level understanding of what the component does to be able to perform step 3. For example, if a component guesses a person's age based on their weight in pounds, the tester would surmise that the most common, or perhaps better phrased as "most probable", inputs would be positive integers (or real) less than 400. The tester may also assume that weights of humans tend to be normally distributed. The tester would specify the mean of the expected input, say 140 pounds in this case, and the deviation from the mean. We generally use 50% of the mean as the deviation, based on experience. Finally, the tester specifies how many test values are desired. If the component accepts only one value as input, the tester may choose to generate 15 values and execute the component 15 times, once with each test value. Or, if the component accepts 25 inputs, the tester may generate 25 values and execute the component only once with these values.

Now we return to step 1. When the tester is undertaking positive testing, the focus is on "most common" or "most expected" data. That is, we tend to select data values that have a high probability of being selected at random from under a probability distribution. If we think of the often cited "grading on a curve" example we see that student grades are normally distributed and that it is most likely that a grade selected at random from the class roster will fall under the bell shaped portion of the distribution. It is not likely that the selected grade will fall under either of the tail areas of the distribution. The same is true for spathic test data generation. If the tester selects positive testing, most probable data are selected, i.e., weighted sampling from under the most dense portions of the probability distribution. When negative testing is selected, we sample under the least dense probability regions, i.e., the tail areas in our prior example.

The name "spathic test data generation" comes from this notion of positive/negative testing for a given probability distribution, mean, deviation, and number of test cases. In effect, we look at the desired number of test cases and we segment or layer the probability distribution into that many layers. If the tester asks for ten test values, we layer the specified input distribution into ten layers. In the future we plan to allow testers to specify the number of layers as well as the number of test values. Note that these subdomains are disjoint. We then see if positive or negative testing has been specified. If positive testing has been specified, we generate ten test values by sampling over our ten-layered probability distribution. We do not necessarily select a value from every layer. We decide how many test values to select from each layer based on the probability density of that layer. If a particular layer has a high probability density, we may select three of the ten values from it. After we decide how many values to select from each layer, we select values from the selected layers and output the values.

For negative testing, we follow the same process but we select more test values from the layers having the least probability density. That is, negative testing results in selection of least likely to occur test values. This is similar to Voas' inverse profile for the system-level extended propagation analysis method [25]. Our approach also

allows a tester to generate additional test cases. Often we may need to continue testing, but we want to follow our original test data generation method and not duplicate the former sampling. We address this by basically splitting the layers or lamella in half and sampling at the center point, depending on the number of additional test values requested. This will be shown in more detail in section 4.2.

4.2 Probability Distributions

The spathic test data generation method is not dependent on a particular probability distribution. It could be used with Weibull distribution, Poisson distribution, exponential distribution, etc. To date, we have investigated the Gaussian distribution. We first present some basic information about the Gaussian distribution.

The Gaussian distribution is also called normal distribution. There are two parameters that define the distribution, the mean and the standard deviation. If the normal distribution has a mean m and a variance s^2, it can be expressed by $N(m,s^2)$. The probability density function for $N(m,s^2)$ is given by

$$f(y) = \frac{1}{\sqrt{2ps^2}} e^{-\frac{(y-m)^2}{2s^2}} \quad (1)$$

If Y is $N(m,s^2)$, then $X=(y-m)/s$ is $N(0,1)$. Here $N(0,1)$ is called the standard normal distribution.

A tenet of this approach is that the input distribution is related to the software's intended function. The tester must select the appropriate distribution to achieve the best results. Guidance for selecting distributions is provided in [21]. If a suitable distribution cannot be used, it is suggested that a small number of Gaussian distributed test values be generated. If these do not appear to be effective, than a different test data technique should be pursued.

4.3 Sampling

This section presents the equations we use to determine the layers and to select test values from the layers. Figure 1 presents a graphical depiction of the test data distribution. We explain the "most probable" or positive case first. Let's assume that we have the standard normal distribution and its deviation is s. The sampling data should meet the following requirements:

1) The data should be in the domain of [-3σ, 3σ]. Here we take 3σ since outside of the domain, the probability is less than 0.003.
2) The interval of the data should have a corresponding distribution. That means that if a point has a large density value, the interval or layer should be small (narrower), otherwise it should be large (wider).

Suppose that the number of samples is $N+1$ in half of the domain, $[0,3s]$, and the data is expressed by $x_0, x_1, x_2,\ldots,x_N$. According to the domain definition, $x_1=0$, and $x_N=3s$. We can define intervals $d_1=x_1-x_0$, $d_2=x_2-x_1,\ldots,d_N=x_N-x_{N-1}$. The d_i $(i=1,2,\ldots,N)$ should comply with the standard normal distribution, which has a deviation s'. The d_i are evenly located on the domain $[0,3s']$. That means $Du=u_{i+1}-u_i$, $u_N=0$, and $u_1=3s'$. So we have $Du=3s'/(N-1)$, and

$$u_i = \Delta u \cdot (N-i) = 3s' \frac{N-i}{N-1} \quad (2)$$

And then from the density function we can get d_i

$$d_i = \frac{1}{\sqrt{2ps'^2}} e^{-\frac{u_i^2}{2s'^2}} \quad (3)$$

The summation of the d_I $(i=1,2,..,N)$ should be $3s$.

Next, we determine the distribution parameter for the new data sampling, $d_1,d_2,\ldots,d_N$. From the preceding, we know that

$$\sum_{i=1}^{N} d_i = 3s \quad (4)$$

Substitute the d_i with equation (3), we get

$$\sum_{i=1}^{N} \left(\frac{1}{\sqrt{2ps'^2}} e^{-\frac{u_i^2}{2s'^2}} \right) = 3s \quad (5)$$

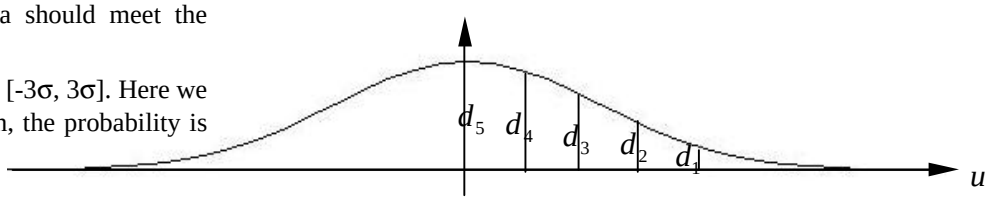

Figure 1. The test data distribution.

Replace u_i with equation (2) in the previous equation, we have

$$\frac{1}{\sqrt{2p}\,s'}\sum_{i=1}^{N} e^{K(N-i)^2} = 3s \qquad (6)$$

Here

$$K = -\frac{9}{2}\cdot\frac{1}{(N-1)^2} \qquad (7)$$

So we can get the new parameter from the equation (6), and

$$s' = \frac{1}{\sqrt{2p}\cdot 3s}\sum_{i=1}^{N} e^{K(N-i)^2} \qquad (8)$$

Since we suppose the original distribution is a standard normal distribution, $s=1$, so the last equation can be written as

$$s' = \frac{1}{3\cdot\sqrt{2p}}\sum_{i=1}^{N} e^{K(N-i)^2} \qquad (9)$$

To generate the least likely values (for negative testing), we perform the same procedure, but $x_i=x_{i-1}+d_{N-i}$, here $i=1, 2,\ldots,N$.

Now assume that we have an operational profile, which satisfies the normal distribution. The distribution parameter is m and s. We want to generate N test values. By using the preceding method, we can get the test sample as follows:

1) Determine the distribution parameter σ' for the sampling data (from equation (8));
2) Calculate the sampling data d_i under the standard normal distribution (from equation (3));
3) Calculate x_i, $x_i=x_{i-1}+d_i$, here i=1, 2,…,N;
4) Inverse the sampling data to the original distribution $y_i=\mu+x_i*\sigma$.

4.4 Additional Sampling

As mentioned in section 4.1, there are situations when we test an application and then realize that additional testing is required and more test data is needed. With random test data generators, we sometimes encounter duplicate test data. We'd prefer to generate additional test data following our original approach, but not duplicate the former sampling.

Suppose the former sampling has N+1 data, and we want to generate another M test values. The previous data are x_0, x_1, …, x_N, and the new data will be y_1, y_2, …, y_M.

(1) If N==M, then we simply insert y_i into the center of x_{i-1}, and x_I, here *(i=1, 2, …,M)*

(2) If N>M, then we insert y_i into the center of x_{i-1}, and x_I, here *(i=1, 2, …,M).*

(3) If N<M, and K_1=M/N, and K_2=M%N (note that % is a modulus operation, so 8%5 equals 3, and 2%5 equals 2), then if k_2=0; we insert K_1 points at every space x_{i-1}, and x_I; If k_2>0; we first insert K_1 point at every space x_{i-1}, and x_I, then for k_2 data, we repeat step (2), but we ensure that the N is changed to $(k_1+1)N$, and that M is replaced by k_2.

4.5 The IntegerSort Example

Our spathic test data generation method has been implemented in Java, but thus far only for the Gaussian distribution. It has been developed in a modular fashion so that other probability distributions can easily be added. To do so, we need only include a new function with the distribution's name in the approach class and insert a command line into our Graphical User Interface (GUI) class. Figure 2 shows the GUI of our spathic test data tool. We found this tool very easy to use.

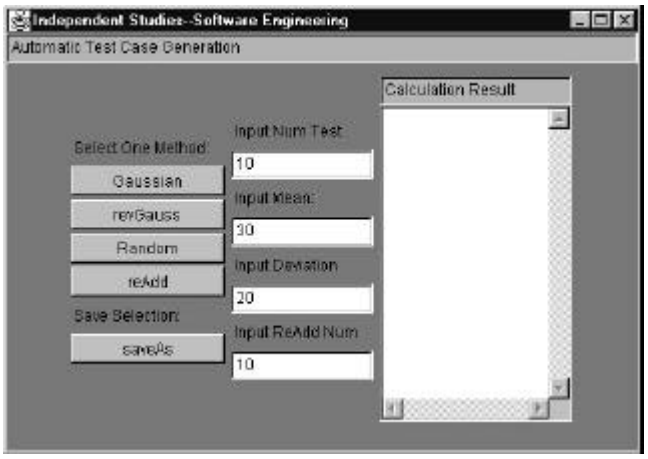

Figure 2. The Spathic Test Data Generation Tool.

We now demonstrate our approach on an example component. IntegerSort is a simple Java component (one method) that accepts a list of integers and sorts them into descending order and outputs the sorted list. As shown in the Appendix, we randomly generated 20 test values for the application. We also generated 20 values using the most probable method (positive testing, also called Gauss method on our tool interface). We specified 0 as the mean and 30,000 as the deviation. We generated 20 values using the least probable method (negative testing, inverse Gaussian, also called revGauss method on our tool interface). The same mean and deviation were used as above.

We used the test coverage analysis tool JCover [8] to analyze statement and branch coverage of this test data. The random data achieved 78.57% branch coverage and 72.73% statement coverage. The spathic test data (positive and negative) achieved 85.71% branch coverage and 77.27% statement coverage. Though these coverage levels are very low and not useful per Frankl and Weiss [24], they are still comparable to the levels achieved by the random data. Next, we used the additional test data capability and generated an additional ten test values for positive testing (Gauss, most probable). We also generated ten additional values for negative testing (revGauss, least probable).

5. Evaluation

An initial case study was carried out to evaluate the method's performance in generating effective test data. To compare fault detection ability of test data we used fault detection effectiveness. Fault detection effectiveness (FDE) is the percentage of seeded faults detected. It is synonymous with average percentage of faults detected (APFD) used by Elbaum, Gable, and Rothermel [4]. We studied the subject programs, the faults introduced into the subject programs, and the test data generation method. A description of the case study follows.

First, nine subject Java programs were selected from publicly available source code repositories and from a repository of graduate student programs. A random test data generator was used to develop tests for each subject program. The tester continued generating values until 80 - 90% branch coverage was achieved (measured per class using JCover) or until it became obvious that additional random values would not increase the coverage. The number of test values was recorded, as were the output results of each test execution (one execution of the component with some number of test values). Second, the spathic test data tool was used to generate the same number of test values. Here, the tester made a decision (based on experience) on the mean of the data and used 50% of the mean as the deviation. Data was generated using Gauss and revGauss. The test values were then also run on the subject program, looking at outputs and branch and statement coverage.

Third, the subject programs were seeded with faults. We based our faults on commonly made Java mistakes [3,13]. Most of the faults were singular syntactic changes (e.g., a relational operator was switched for another one), small syntactic faults [10]. We did not examine their semantic size. The test values for most probable and for least probable cases were then executed on the faulty versions. The results were compared to the results from testing the unmodified subject program. If the output of the faulty version did not match the output of the original subject program, the fault was said to be detected. Fault detection effectiveness was measured as the percentage of faults that were detected by a method. So for example, if the spathic most probable method (Gauss) found four of eight faults, it is said to have an FDE of 50%.

The study is detailed in Table 1. The first column shows the subject programs used in the case study. Weight processes a list of human weights. Score processes test scores. IntegerSort was described in Section 4. LetterGrade processes grades and outputs a report. Calculate2 takes three numeric inputs and calculates the sum and product. LargeSmall takes five numeric inputs and outputs the largest and smallest values. Perfect takes numerical input and indicates if the number is deficient or abundant. SortArray sorts a given list of values into ascending or descending order. IntegerStats calculates the mean and standard deviation of a list of values. The programs varied from 73 to 284 lines of code. Recall that we are interested in applying input probability distribution testing at the unit level, so we tried to use programs that were "unit" sized.

The second column shows the number of faulty versions of each subject program. One fault was seeded in each version. The third column indicates how many random test data sets were used for the subject program, with the number of test values in each set given in square brackets. For example, we executed 40 random method test sets on "buggy" versions of Weight (ten test sets per fault-seeded version) with seven test values in each test set. The fourth and fifth columns provide the same information for the spathic Gauss and spathic revGauss methods.

Table 1. Study design.

Subject Program	Number of Faulty Versions	Number of Random Test Sets [Number of test values in each set]	Number of Spathic Gauss Test Sets [Number of test values in each set]	Number of Spathic revGauss Test Sets [Number of test values in each set]
Weight	4	40 [7]	4 [7]	4 [7]
Score	4	40 [7]	4 [7]	4 [7]
IntegerSort	6	60 [20]	6 [20]	6 [20]
LetterGrade	4	40 [10]	4 [10]	4 [10]
Calculate2	8	8 [3]	8 [3]	8 [3]
LargeSmall	3	3 [5]	3 [5]	3 [5]
Perfect	3	3 [7]	3 [7]	3 [7]
SortArray	5	5 [20]	5 [20]	5 [20]
IntegerStats	3	3 [20]	3 [20]	3 [20]

The results are detailed in Tables 2 and 3. The first column lists the subject program. The second column gives the fault detection effectiveness of the random data sets. The third and fourth columns provide the same information for the spathic Gauss and spathic revGauss test sets.

In Table 2, it should be noted (from Table 1) that for faulty versions where spathic data outperformed random data, we generated many more sets of random test values than of spathic Gauss or revGauss to ensure conservative results. That is, ten times more random test values may have been needed to achieve the shown fault detection levels. For example, for the LetterGrade program, we generated ten sets of ten test values for each fault-seeded version. Of those ten sets, only three of the ten sets found bug1, two out of ten found bug2, and four out of ten found bug3.

Table 3 shows the average branch and statement coverage achieved by the test sets. Column 1 lists the subject program. Column 2 lists the average branch coverage of the random data, with the statement coverage average in square brackets. For example, SortArray random data achieved, on average, 81% branch coverage and 88% statement coverage. The third and fourth columns provide the same information but for spathic Gauss and spathic revGauss test sets. Upon examining the code, we noted that the programs with low coverage had numerous error checking branches that our test data would never execute. For example, "report error if no input values are specified."

Table 2. Study results - FDE.

Subject Program	**Percent of Faults Detected (FDE)**		
	Random Test Data	Gauss Test Data	revGauss Test Data
Weight	0	75	0
Score	0	75	75
IntegerSort	60	60	60
LetterGrade	25	50	50
Calculate2	66.67	66.67	66.67
LargeSmall	33.33	33.33	33.33
Perfect	33.33	33.33	33.33
SortArray	20	20	20
IntegerStats	100	100	100

Informally, we also looked at the performance of generated "additional test data." For example, for the program Weight we generated five additional values using all three methods. The revGauss values found 50% of the bugs, while the other two methods found none. For LetterGrade, all three methods found 25% of the faults when five additional values were generated for each.

Table 3. Study results – coverage.

Subject Program	**Average Branch Coverage [Average Statement Coverage]**		
	Random Test Data Sets	Gauss Test Data Sets	revGauss Test Data Sets
Weight	50 [48]	60 [52]	50 [48]
Score	50 [38]	50 [38]	50 [38]
IntegerSort	79 [73]	86 [77]	86 [77]
LetterGrade	79 [71]	86 [76]	79 [71]
Calculate2	50[93]	50[93]	50[93]
LargeSmall	50[77]	50[87]	50[87]
Perfect	77[77]	83[80]	83[80]
SortArray	81[88]	81[88]	81[88]
IntegerStats	72[78]	72[78]	72[78]

As can be seen, the spathic test data generation method found more faults than the random test data, while achieving higher branch and statement coverage (though not impressive coverage levels). The test sets were the same size with the exception of some random test sets that were larger (to ensure that bias was in favor of the control method). It must also be noted that many faults went undetected by all three testing methods. We found the method to be very easy to use. It did require us to determine the number of test cases desired, but this is an activity already routinely performed by practitioners.

The initial study is viewed as a proof of principle test, i.e., does the spathic method have sufficient potential to warrant additional testing? The initial results are promising and additional testing is warranted. The results, however, are limited and the effectiveness of the method on a broader scale remains to be seen.

Though this was a small study, we did consider threats to its validity. Internal validity threats deal with the causal relationship between the independent and dependent variables. We attempted to limit the threat by validating the tools and processes we used for data collection. For example, we used a commercially available coverage tool and we validated the performance of the spathic test data generation tool before undertaking the study. A major threat to external validity (generalization of results) for our study is the representativeness of our subject programs and faults as well as our small sample size. We attempted to

control this threat by selecting programs from Internet repositories and selecting sample faults from lists of common Java errors. There is also the threat of lack of construct validity (are the measures appropriate?). The dependent variable FDE does not account for fault severity, fault "hardness" (how difficult to detect), etc.

These are initial results based on a small sample. Further study and research is required before any generalizations or broad conclusions can be reached. However, the initial results encourage us that this technique may have merit.

6. Conclusions And Future Work

In this paper, we have proposed a method for generating test data and presented preliminary data to support this method. The proposed method is applicable to unit level testing and requires less effort on the part of the tester than many specification-based or structural criteria-based techniques. The approach also supports the generation of additional non-duplicative test data later in the testing cycle.

Our initial evaluation of the proposed method shows that the generated spathic test data achieves similar or higher levels of branch and statement coverage, and that it has higher fault detection effectiveness (FDE) than randomly generated test data. We believe the approach works well with either small or large input ranges, when a small amount of white box testing information is available. If a tester has some knowledge of the most likely problem areas or if this is self-evident from the type of application or component, the approach works well. The white box information may be as simple as "it is most likely that negative integers will detect faults." As noted above, the initial results are promising but very limited in scope. A much larger scale study with a variety of distribution functions is required before any broad conclusions can be reached.

7. Acknowledgments

Our thanks to Naresh Mohamed for generating, executing, and assisting with analysis of the tests. We thank Tina Gao and Bill Kidwell for their timely assistance. Thanks to Professors Jeff Offutt and Dave Huffman for their helpful comments. Our appreciation to Man Machine Systems and their kind donation of JCover for use in our research program.

8. References

[1] Ammann, P., and Offutt, J. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance, Safety, Security, Reliability, Fault Tolerance, Concurrency and Real Time (COMPASS '94)*, 1994, 69 –79.

[2] Cochran, W. Sampling Techniques. Wiley, 1977.

[3] Common Java Syntax Errors. http://www.open.ac.uk/StudentWeb/m874/!synterr.htm.

[4] Elbaum, S., Gable, D., and Rothermel, G. Understanding and measuring the sources of variation in the prioritization of regression test suites. IEEE, *Proceedings of the Seventh International Software Metrics Symposium, METRICS 2001*, 169-179.

[5] Hamlet, D. and Voas, G. Faults on its sleeve, amplifying software reliability testing. In *Proceedings, 1993 International Symposium on Software Testing and Analysis* (Cambridge, MA, June). ACM, 89-98.

[6] Howden, W.E. Functional Program Testing and Analysis. McGraw-Hill, New York, 1987.

[7] Howden, W.E.,and Huang, Y. Software trustability analysis. *ACM Transactions on Software Engineering and Methodology*, Volume 4, Number 1, January 1995, 36-64.

[8] Code Coverage Analyzer for Java - JCover, http://www.mmsindia.com/JCover.html.

[9] Ntafos, S. On comparisons of random, partition, and proportionalpPartitiontTesting. *IEEE Transactions on Software Engineering*, Volume 27, Number 10, 949-960.

[10] Offutt, J. and Hayes, J. Huffman. A semantic model of program faults. *The Proceedings of the International Symposium on Software Testing and Analysis*. ACM, San Diego, California, January 1996, 195-200.

[11] Ostrand, T.J. and Balcer, M.J. The category-partition method for specifying and generating functional test. *Communications of the ACM*, Volume 31, Number 6, June 1988, 676-686.

[12] Podurski, A., Masri, W., McCleese, Y., and Wolff, F. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, Volume 8, Number 3, July 1999, 262-283.

[13] Reilly, D. Top ten errors Java programmers make. http://www.javacoffeebreak.com/articles/toptenerrors.html.

[14] Zhu, H., Hall, P.A.V., May, J.H.R. Software unit test coverage and adequacy. *ACM Computing Surveys*, Volume 29, Number 4, December 1997, 366-427.

[15] Musa, J.D. The operational profile in software reliability engineering: an overview. *Proceedings of Third International Symposium on Software Reliability Engineering*, 1992, 140 –154.

[16] Musa, J. D. Operational profiles in software reliability engineering. *IEEE Software Magazine*, volume 10, number 2, March 1993, pp. 14-32.

[17] Thévenod--Fosse, P. and Waeselynck, H. An investigation of statistical software testing. *Journal of Software Testing, Verification and Reliability*, 1(2), 1991, 5-25.

[18] Sahinoglu, M., von Mayrhauser, A., Hajjar, A., Chen, T., Anderson, C. On the efficiency of a compound poisson stopping rule for mixed strategy testing, *Procs. IEEE Aerospace Applications Conference* 1999, March 1999, Snowmass, CO.

[19] Howden, W.E.: Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*. SE-2, 208-215, 1976.

[20] Woit, D. A framework for reliability estimation. *Proc. 5th IEEE International Symposium on Software Reliability Engineering (ISSRE'94)*, November 6-9, 1994. pp. 18-24.

[21] Hayes, J. Huffman, and Gao, H. Fault detection effectiveness of spathic test data generated according to a geometric function. University of Kentucky Computer Science Department Technical Report TR 344-02, May 2002.

[22] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, pages 3-12, 1993.

[23] Littlewood, B., and Wright, D. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Transactions on Software Engineering*, 23(11): 673-683, 1997.

[24] Frankl, P., and Weiss, S. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19, 8 (1993), 774-787.

[25] Voas, J., Charron, F., McGraw, G., Miller, K., Friedman, M Predicting how badly "good" software can behave. *IEEE Software*. Volume 14, Issue 4 , July-Aug. 1997, 73–83.

[26] Fujiwara, S., Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. 1991. Test selection based on finite state models. *IEEE Transactions on. Software Engineering*, *SE-17*, 6 (June), 591–603.

[27] Hall,P.A.V.And Hierons, R. 1991. Formal methods and testing. Tech. Rep. 91/16, Dept. of Computing, The Open University.

[28] Ural,H.And Yang, B. 1993. Modeling software for accurate data flow representation. In *Proceedings of 15th International Conference on Software Engineering* (May), 277–286

Appendix

Initial Values			Additional Values	
Random	Gauss	RevGauss	Gauss	RevGauss
0 -87747	0 -89894	0 -89894	10 addt'l	10 addt'l
1 -84074	1 -66485	1 -89622	0 -78189	0 -89758
2 -70291	2 -46032	2 -88982	1 -56258	1 -89302
3 -65583	3 -29700	3 -87608	2 -37866	2 -88295
4 -64117	4 -17781	4 -84908	3 -23740	3 -86258
5 -63983	5 9831	5 -80062	4 -13806	4 -82485
6 -63606	6 4985	6 -72112	5 -7408	5 -76087
7 -55221	7 2286	7 -60194	6 -3636	6 -66153
8 -30224	8 -911	8 -43862	7 -1598	7 -52028
9 -22852	9 -272	9 -23409	8 -591	8 -33635
10 4913	10 0	10 0	9 -136	9 -11704
11 7266	11 272	11 23409		
12 35937	12 911	12 43862		
13 36928	13 2286	13 60194		
14 58397	14 4985	14 72112		
15 62578	15 9831	15 80062		
16 74871	16 17781	16 84908		
17 76147	17 29700	17 87608		
18 81743	18 46032	18 88982		
19 84565	19 66485	19 89622		

Under Initial Values, Column 1 shows the 20 randomly generated test values. Column 2 shows the 20 test values generated using the most probable method (called Gauss method on our tool interface) – the mean was 0 and the deviation was 30,000. Column 3 shows the 20 values generated using the least probable method (inverse Gaussian, also called revGauss method on our tool interface) – we used the same mean and deviation as above. Under Additional Values, we generated 10 additional test values using the Gauss and RevGauss methods.

.