RELIABILITY VIA DESIGNED EXPERIMENTS

The theory of experimental design was developed in response to the fact that experimental results are inherently variable. In fields such as physics and chemistry, this variability is often quite small, and, for experiments conducted in a classroom environment, it is not unusual to think of the "correct" result for an experiment. Even in the classroom, however, experience indicates that the results are variable, with the variability arising from complexities of the measurement procedure as well as from the inherent variability of the experimental material. The precision of an experiment, and therefore the statistical design of the experiment, becomes increasingly important as the complexity of the experiment increases and the differences between experimental results grow relatively smaller.

One area in which statistical design of experiments is important is in the design of experiments for reliability. For instance, in a complex manufacturing process, such as that used in the production of very large scale integration (VLSI) components, we wish to know how measurable characteristics of the manufacturing process affect the reliability of the product. We can determine these effects by taking periodic samples of the product and experimentally determining their expected lifetimes. Because of the high precision required in measuring a complicated manufacturing process as well as determining the success or failure of a complex item, such as a VLSI circuit, it is important to design the experiment so as to properly take into account all factors affecting an item's reliability as well as to accurately and precisely measure the effects of the factors.

RELIABILITY THEORY OVERVIEW

Reliability is the probability that a system will operate without failure for a specified period of time (the design life) in a specified environment (e.g., ambient temperature, power supply voltage, energetic particle flux). This is simply the cumulative probability distribution (CDF) of success. We may consider reliability to be a measure of the system's success in performing its intended function. For example, suppose that 1000 identical electronic parts are tested in the environment in which they are expected to operate. During an interval of time $(t - \Delta t, t)$, we observe that 97 of the original 1000 components have failed. Since reliability is the CDF of success, the reliability at time t, $R(t)$, is

$$\begin{aligned} R(t) &= \frac{\text{number of components surviving at time } t}{\text{total number of components under test}} \\ &= \frac{903}{1000} = 0.903 \end{aligned} \tag{1}$$

If $\mathbf{t}$ is a random variable denoting the time to failure, a system's reliability function at time t is given by

$$R(t) = P(\mathbf{t} > t) \tag{2}$$

The CDF of failure, $F(t)$, is the complement of $R(t)$

$$R(t) = 1 - F(t) \tag{3}$$

If the probability density function (pdf) associated with the random variable $\mathbf{t}$ is given by $f(t)$, we can then rewrite $R(t)$, given by Eq. (3), as follows:

$$R(t) = 1 - \int_0^t f(x)\,dx \tag{4}$$

If we take the time derivative of Eq. 4, we obtain the following relationship between and $R(t)$ and $f(t)$:

$$\frac{dR(t)}{dt} = -f(t) \tag{5}$$

For example, suppose that $f(t)$ is exponential with parameter λ. The pdf for a model of accelerated life testing, the exponential distribution model, has this form. In this case, $f(t)$ =

$\lambda e^{-\lambda t}$, and the reliability function $R(t)$ is

$$R(t) = 1 - \int_0^t \lambda e^{-\lambda x}\, dx = e^{-\lambda t} \tag{6}$$

We are now in a position to express the probability of a system's failing in a time interval $[t_1, t_2]$ in terms of its reliability function:

$$P(t_1 \le \mathbf{t} \le t_2) = \int_{t_1}^{t_2} f(x)\, dx = R(t_1) - R(t_2) \tag{7}$$

The failure rate in the interval $[t_1, t_2]$ is defined as the probability that a failure per unit time occurs in this interval, given that no failure has occurred prior to t_1. The failure rate is given by

$$\frac{R(t_1) - R(t_2)}{(t_2 - t_1)R(t_1)} \tag{8}$$

If we now replace t_1 by t and t_2 by $t + \Delta t$, we can rewrite the failure rate as

$$\frac{R(t) - R(t + \Delta t)}{\Delta t R(t)} \tag{9}$$

The hazard function $h(t)$, or instantaneous failure rate, is defined as the limit of the failure rate given in Eq. (9) as Δt approaches zero,

$$h(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)} = \frac{1}{R(t)}\left[-\frac{d}{dt}R(t)\right] = \frac{f(t)}{R(t)} \tag{10}$$

From Eqs. (5) and (10), we obtain the following relationship between the reliability function $R(t)$ and the hazard function $h(t)$:

$$R(t) = e^{-\int_0^t h(x)\, dx} \tag{11}$$

The key equations relating $R(t)$, $h(t)$, $F(t)$, and $f(t)$ are Eqs. (5), (10), and (11).

We now give a simple example to show how failure data can be used to estimate the hazard rate and reliability. Suppose a light bulb manufacturer is interested in estimating the mean life of the bulbs. Five hundred bulbs are tested under the same conditions under which they are expected to be used by the firm's customers. The bulbs are observed during the test; the number of failures observed in nonoverlapping 1000 h intervals is shown in Table 1. We now wish to plot the following quantities:

Table 1. Number of Failed Light Bulbs per Time Interval

Hours in the Time Interval	Failures Observed
0–1000	237
1001–2000	73
2001–3000	53
3001–4000	34
4001–5000	32
5001–6000	27
6001–7000	24
7001–8000	20
Total failures	500

- $\tilde{f}(t)$, *the Failure Density Function Estimated from the Data.* We compute this as:

$$\tilde{f}(t) = \frac{n_f(t)}{n_0 \Delta t} \tag{12}$$

  where $n_f(t)$ is the number of light bulbs that have failed by time t, Δt is the length of the interval (1000 h for this example), and n_o is the total number of light bulbs being tested (500 for this example).
- $\tilde{h}(t)$, *the Hazard Function Estimated from the Data.* This is computed as:

$$\tilde{h}(t) = \frac{n_f(t)}{n_s(t)\Delta t} \tag{13}$$

  where $n_s(t)$ is the number of light bulbs surviving at time t.
- $\tilde{R}(t)$, *the Reliability Function Estimated from the Data.* From Eq. (10), we can write

$$\tilde{R}(t) = \frac{\tilde{f}(t)}{\tilde{h}(t)} \tag{14}$$

- $\tilde{F}(t)$, *the CDF of Failure Estimated from the Data.* Since the CDF of failure is the complement of the reliability function, we can write

$$\tilde{F}(t) = 1 - \tilde{R}(t) \tag{15}$$

The computations for each of these quantities are shown in Table 2.

FACTORS AFFECTING RELIABILITY

For many physical systems, the physics of the failure mechanisms for those systems can be used to estimate their reliability. There are many mechanisms for the failure of electrical and electronic devices. For instance, the time to failure of integrated circuits due to electromigration is affected by two factors—the current density through the circuit and the circuit's temperature. The rate at which corrosion that can deteriorate the leads outside of a packaged integrated circuit and thereby cause its failure is determined by the relative humidity of the circuit's operating environment. For thin-film integrated circuit resistors, the rate at which their resistance changes over time is affected by the temperature of their operating environment. The interested reader may refer to Elsayed (1) for more details.

In each of these cases, factors affecting a system's reliability are either physical properties of the system itself, characteristics of the process by which it was manufactured, or characteristics of its operating environment that can be measured. Experiments can be designed to determine the effects of these factors on the system's reliability. Such experiments must be designed with the following issues in mind:

- How to take into account similarities and differences between individual experimental units in the design.
- How to estimate the effects of individual factors that are believed to determine a unit's reliability.
- How to identify and model interactions between factors.

Table 2. Computing $\tilde{f}(t)$, $\tilde{h}(t)$, $\tilde{F}(t)$, and $\tilde{R}(t)$

Time Interval (Hours)	Estimated Failure Density	Estimated Hazard Function	Estimated Failure CDF	Estimated Reliability Function
0–1000	$\frac{237}{500 \times 10^3} = 4.74\text{E} - 04$	$\frac{237}{500 \times 10^3} = 4.74\text{E} - 04$	0.00E + 00	1.00E + 00
1001–2000	$\frac{73}{500 \times 10^3} = 1.46\text{E} - 04$	$\frac{73}{263 \times 10^3} = 2.78\text{E} - 04$	4.74E − 01	5.26E − 01
2001–3000	$\frac{53}{500 \times 10^3} = 1.06\text{E} - 04$	$\frac{53}{190 \times 10^3} = 2.79\text{E} - 04$	6.20E − 01	3.80E − 01
3001–4000	$\frac{34}{500 \times 10^3} = 6.80\text{E} - 05$	$\frac{34}{137 \times 10^3} = 2.48\text{E} - 04$	7.26E − 01	2.74E − 01
4001–5000	$\frac{32}{500 \times 10^3} = 6.40\text{E} - 05$	$\frac{32}{103 \times 10^3} = 3.11\text{E} - 04$	7.94E − 01	2.06E − 01
5001–6000	$\frac{27}{500 \times 10^3} = 5.40\text{E} - 05$	$\frac{27}{71 \times 10^3} = 3.80\text{E} - 04$	8.58E − 01	1.42E − 01
6001–7000	$\frac{24}{500 \times 10^3} = 4.80\text{E} - 05$	$\frac{24}{44 \times 10^3} = 5.45\text{E} - 04$	9.12E − 01	8.80E − 02
7001–8000	$\frac{20}{500 \times 10^3} = 4.00\text{E} - 05$	$\frac{20}{20 \times 10^3} = 1.00\text{E} - 03$	9.60E − 01	4.00E − 02

TYPES OF EXPERIMENTAL DESIGN

Randomized Block Designs

When setting up an experiment to compare different treatments, each treatment must be applied to several units—if we were to apply a treatment to only one unit, we would not be able to determine whether differences in the responses from two units were caused by differences in the treatments or whether they were due to the units being inherently different. The simplest type of experiment to compare the effects of n treatments is one in which the first treatment is applied to x_1 units, the second treatment to x_2 units, and so forth until the nth treatment is applied to x_n units. The only recognizable difference between the units in this type of experiment is the treatment that is applied. This design is called a completely randomized design.

Even though a treatment is applied to multiple units in a randomized design, the ambiguity that occurs if only a single unit is treated in a particular manner is not eliminated. It may be the case that a treatment might be applied to all of the units which respond in a particular fashion. However, if the experimenter has any ideas of which units are most likely to behave similarly, they can be used to control the allocation of treatments to units. The idea is that the experimental units are grouped into blocks that are "similar"; each such block will then include roughly equal numbers of units for each treatment. This control is referred to as blocking, and the resulting design is the randomized block design. This type of experiment might be appropriate for determining the effects of changing one step in the manufacturing process for integrated circuits. The measured change to the fabrication process would be the treatment in this case. The wafers being sampled from the production line might be divided into blocks; the blocks are characterized by their distance from the center of the wafer.

In the randomized and randomized block designs, we assume that each experimental unit has an inherent yield that is modified by the effect of the treatment to which it is subjected. For the randomized design, we can express this by

$$y_{jk} = \mu + t_j + \epsilon_{jk} \tag{16}$$

where

y_{jk} is the yield for unit k after the application of treatment j
μ is an average yield for all of the experimental units
t_j is the deviation of all of the units that have undergone treatment j from the average of all the treatments included in the experiment.
ϵ_{jk} is represents the deviation of unit k, having undergone treatment j, from the average yield μ

For the randomized block experiment, we need to take into account the differences between blocks. This is done by modifying Eq. (16) as follows:

$$y_{ij} = \mu + b_i + t_j + \epsilon_{ij} \tag{17}$$

The average yield, μ, is the same as for the randomized experiment. The yield, however, is now the yield of treatment j in the ith block, y_{ij}. The additional term b_i represents the average deviation from μ of the units in the ith block. Finally, ϵ_{ij} represents the deviation of the units in the ith block that have undergone treatment j from the average yield of all of the units in the ith block, $\mu + b_i$.

Most experimental data is examined using the *analysis of variance* technique. Detailed treatments of this analysis technique may be found in Refs. (2) and (3). This analysis has two functions:

1. It divides the total variation between the experimental units into components that represent the different sources of variation. This provides a way of assessing the relative importance of those sources.
2. It provides estimates of the underlying variation between the units themselves, which can be used in reasoning about the effects of the treatments.

The analysis of variance for the randomized block experiment takes into account the following sources of variation:

Variations between blocks
Variations between different treatments
Variations due to inconsistency of treatment differences over the different blocks

The analysis of variance relationship is given by

$$\begin{aligned}\sum_{ij}(y_{ij}-y_{\bullet\bullet})^2 &\equiv \sum_{ij}(y_{i\bullet}-y_{\bullet\bullet})^2\\ \text{Total Sum of Squares} &\equiv \text{Block SS}\\ &+\sum_{ij}(y_{\bullet j}-y_{\bullet\bullet})^2+\sum_{ij}(y_{ij}-y_{i\bullet}-y_{\bullet j}+y_{\bullet\bullet})^2\\ &+\text{Treatment SS}+\text{Error SS}\end{aligned} \tag{18}$$

The use of a dot instead of a suffix in Eq. (18), such as $y_{\bullet j}$, indicates the mean value of y over all possible values of the suffix j. The relative magnitudes of between-block variance, between-treatment variance, and variation between the units are obtained by comparing the block, treatment, and error mean squares. If the number of blocks is b, and the number of treatments is t, the block, treatment, and error mean squares are as follows:

$$\begin{aligned}\text{Block mean square} &= \sum_{ij}(y_{i\bullet}-y_{\bullet\bullet})^2\Big/(b-1)\\ \text{Treatment mean square} &= \sum_{ij}(y_{\bullet j}-y_{\bullet\bullet})^2\Big/(t-1)\\ \text{Error mean square} &= \sum_{ij}(y_{ij}-y_{i\bullet}-y_{\bullet j}\\ &\qquad +y_{\bullet\bullet})^2\Big/(b-1)(t-1)\end{aligned} \tag{19}$$

Each of the divisors in the above definitions of mean squares are referred to as *degrees of freedom* for that source of variance. The mean squares may also be compared using F tests if we assume that the variation of the experimental units follows a normal distribution.

Factorial Designs

Factorial designs are employed when we wish to examine the interactions among various factors affecting the results of an experiment. A *factor* is a set of treatments that can be applied to experimental units. For instance, in an experiment on metal fractures, one factor might be the thickness of the material. A factor's *level* is a specific treatment from the set of treatments that make up the factor. For example, there might be three different thicknesses of the material being investigated in an experiment on metal fractures. Each thickness would constitute a different level of the thickness factor. An *experimental treatment* is the description of the way in which a particular unit is treated; the treatment comprises one level from each factor.

Consider a simple example in which a manufacturer of light bulbs is considering changing the filament material and the shape of the filament with the goal of increasing the lifetime of the bulbs. In addition to the current shape of the filament S_1, there are three possible new shapes (S_2, S_3, and S_4), and the manufacturer wishes to determine the effects of the new material and the new shapes on the expected lifetime of the bulbs. For this experiment, there would be two factors: filament material F and shape S. There are two levels of the filament material factor (current material C, new material N) and four levels of the shape factor (current shape S_1, S_2, S_3, and S_4). The set of experimental treatments is given by $F \times S$, that is:

$$\{(C, S_1), (C, S_2), (C, S_3), (C, S_4), (N, S_1), (N, S_2), (N, S_3), (N, S_4)\}$$

A total of 800 bulbs are included in the experiment; 400 are constructed using filaments made of the proposed new material, and 400 are constructed using the material currently used by the firm in the bulbs it distributes commercially. The failure density, hazard function, reliability function, and CDF of failure are obtained as shown in the example in the first section. The expected lifetime of the bulbs under each experimental treatment is computed from $\tilde{R}(t)$ for that treatment as the sum $\sum_{i=1}^{N} \Delta t_i \tilde{R}(t_i)$, where i denotes successive test intervals, Δt_i denotes the length of interval i, N denotes the interval at the conclusion of which no bulbs were functioning, and $\tilde{R}(t_i)$ denotes the estimated reliability function at the end of the ith test interval. The results of the experiment are given in Table 3. Each entry in the table gives the observed expected lifetime (measured in hours) of the bulbs under that particular experimental treatment. There are several ways in which we can interpret these results:

1. We could consider the effects of changing the shape of the filament for each type of filament material. For the bulbs using the current filament material, the effect of changing the shapes from S_1 to S_2, S_2 to S_3, or S_3 to S_4 increases the expected lifetime at each change in shape, with the change from S_3 to S_4 having the largest effect. A similar increase in expected lifetime is for the bulbs using filaments made of the new material as the shape of the filament is changed. For the bulbs using the new material in their filaments, the largest increase in the expected lifetime is associated with changing from S_3 to S_4.
2. We could consider the expected lifetime differences between experimental and current filaments for each of the four shapes. These are 17.1 h for shape S_1, 20.8 h for S_2, 22.8 h for S_3, and 29.2 h for S_4. From this viewpoint, we see that the experimental material has an advantage for each of the four shapes, and has the biggest advantage for shape S_4.

Table 3. Observed Expected Lifetimes of Experimental versus Current Bulbs

	Filament Shape			
Filament Type	Current (S_1)	S_2	S_3	S_4
Current material	1895.20	1908.70	1922.30	1939.40
Experimental material	1912.30	1929.50	1945.10	1968.60

3. We can consider first the average difference between the two types of filaments, which for our example is 22.48 h. Secondly, we can consider the average response to shaping the filaments in different ways; in this case 1903.75, 1919.19, 1938.70, and 1954.00, and then the way in which the overall pattern differs from a combination of these two effects. We can express the way in which the overall pattern differs from a combination of two effects by saying either that the difference in expected lifetime between the current and experimental filaments is largest for shape S_4 or that the increase in expected lifetime in response to changing the filament shape from S_1 to S_4 is larger for the experimental filament material than for the material currently in use.

We can express this third approach in the following model:

$$t_{jk} = f_j + s_k + (fs)_{jk} \tag{20}$$

where t_{jk} is the treatment effect for material j and shape k, f_j is the average treatment effect for material j, s_k is the average treatment effect for shape k, and $(fs)_{jk}$ is the difference between t_{jk} and $f_j + s_k$. Effects involving comparisons between levels of only one factor are called *main effects* of that factor, while those effects involving comparisons for more than a single factor are called *interactions.* We can define these effects more precisely as follows.

The main effects of a factor is a comparison between the expected yields for different levels of one factor, averaging over all levels of all the other factors. We can write this as

$$\sum_j l_j t_{j\bullet} \tag{21}$$

where $\sum_j l_j$ is 0 and $t_{j\bullet}$ represents the average value of t_{jk} over all possible levels of factor k. We may write $t_{j\bullet} = \sum_k t_{jk}/n_j$. The interaction between two factors is written as

$$\sum_k m_k \sum_j l_j t_{jk} \tag{22}$$

where $\sum_k m_k$ is 0.

Returning to Eq. (20), we can recognize f_j, s_k, and $(fs)_{jk}$ as main effects and interactions if we define them as follows:

$$\begin{aligned} f_j &= t_{j\bullet} - t_{\bullet\bullet} \\ s_k &= t_{\bullet k} - t_{\bullet\bullet} \\ (fs)_{jk} &= (t_{jk} - t_{j\bullet}) - (t_{\bullet k} - t_{\bullet\bullet}) \\ &= (t_{jk} - t_{\bullet k}) - (t_{j\bullet} - t_{\bullet\bullet}) \end{aligned}$$

For the numerical values used in the example above, the treatment effects t_{jk} are estimated by the deviations of treatment yields from the overall average of 1927.64. These deviations are shown in Table 4. The estimates of the effects are given below:

Table 4. Deviations of Treatment Yields from Overall Average

	Filament Shape			
Filament Type	Current (S_1)	S_2	S_3	S_4
Current material	−32.44	−18.94	−5.34	11.76
Experimental material	−15.34	1.86	17.46	40.96

$$\begin{aligned} f_1 &= (-32.44 - 18.94 - 5.34 + 11.76)/4 = -11.24 \\ f_2 &= (-15.34 + 1.86 + 17.46 + 40.96)/4 = 11.24 \\ s_1 &= (-32.44 - 15.34)/2 = -23.89 \\ s_2 &= (-18.94 + 1.86)/2 = -8.54 \\ s_3 &= (-5.34 + 17.46)/2 = 6.06 \\ s_4 &= (11.76 + 40.96)/2 = 26.36 \\ (fs_{11}) &= (-32.44) - (-11.24) - (-23.89) = 2.69 \\ (fs_{12}) &= (-18.94) - (-11.24) - (-8.54) = 0.84 \\ (fs_{13}) &= (-5.34) - (-11.24) - (6.06) = -0.16 \\ (fs_{14}) &= (11.76) - (-11.24) - (26.36) = -3.36 \\ (fs_{21}) &= (-15.34) - (11.24) - (-23.89) = -2.69 \\ (fs_{22}) &= (1.86) - (11.24) - (-8.54) = -0.84 \\ (fs_{23}) &= (17.46) - (11.24) - (6.06) = 0.16 \\ (fs_{24}) &= (40.96) - (11.24) - (26.36) = 3.36 \end{aligned}$$

The results above confirm the qualitative conclusions stated earlier. The average difference in expected lifetime between light bulbs having filaments made of different materials is 22.48 h. The major effect of changing shapes from S_1 to S_1, S_2 to S_3, and S_3 to S_4 is that the difference in expected life between bulbs using the current material and those using the new material increases. The interaction pattern occurs when changing from S_1 to S_2 and from S_3 to S_4. Using the new material and changing from S_1 to S_2 produces a additional positive difference of $fs_{22} - fs_{21} = 1.85$ h to add to the main effect difference, $S_2 - S_1 = 15.35$ h, for a total effect of 17.20. Using the current material and changing from S_1 to S_2 produces a negative difference of $fs_{12} - fs_{11} = -1.85$ h to add to the main effect difference for a total effect of 13.50 h. Using the new material and changing from S_3 to S_4 produces an additional positive difference of $fs_{24} - fs_{23} = 3.20$ h to add to the main effect difference, $S_4 - S_3 = 20.30$ h, for a total effect of 23.50. Using the current material and changing from S_1 to S_2 produces a negative difference of $fs_{14} - fs_{13} = -3.20$ h to add to the main effect difference for a total effect of 17.10 h.

One of the advantages of using a factorial structure in experimental design is that we are able to examine interactions between factors, as illustrated above. There are two additional advantages. First, conclusions about the effects of a factor have a broader validity because of the range of conditions under which that factor has been studied. Second, and even more important, is that a factorial experiment essentially allows several experiments to be done simultaneously. We can illustrate this advantage with the following example. Suppose that we want to investigate the effects of three factors, each at two levels, and that we only have enough resources for 24 observations. The factors are X, Y, and Z with levels x_0 and x_1, y_0 and y_1, z_0 and z_1. There are three designs that we consider:

1. We can have three separate experiments, one for each factor, with 8 observations per experiment:
   - ($x_0y_0z_0$, $x_1y_0z_0$), four observations each
   - ($x_0y_0z_0$, $x_0y_1z_0$), four observations each
   - ($x_0y_0z_0$, $x_0y_0z_1$), four observations each

In this set of three experiments, we isolate the effect of each factor in turn by controlling all other factors. This is considered to be the classical scientific experiment.

2. We can reduce the resources wasted in the first experiment by using $(x_0y_0z_0)$ in each of the three individual experiments. Instead, the four distinct treatments may be replicated equally as follows:

$$(x_0y_0z_0),\ (x_1y_0z_0),\ (x_0y_1z_0),\ (x_0y_0z_1)$$

with six observations each.

3. We can design a factorial experiment with the following eight treatments:

$$(x_0y_0z_0),\ (x_0y_0z_1),\ (x_0y_1z_0),\ (x_0y_1z_1),\ (x_1y_0z_0),\ (x_1y_0z_1),\ (x_1y_1z_0),\ (x_1y_1z_1)$$

Each treatment in this experiment would have three observations.

To compare the three designs, we can look at the variance of the comparison of mean yields for x_0 and x_1 (the comparisons of the mean yields for y_0 and y_1 and z_0 and z_1 will be equivalent). The three experiments give the following variances for $X_1 - X_0$:

1. $2\sigma^2/4$ for the classical scientific experiment
2. $2\sigma^2/6$ for the equal replication of distinct treatments
3. $2\sigma^2/12$ for the factorial experiment

The factorial experiment gives the smallest variance for $x_1 - x_0$, and is therefore the most efficient of the three experiments—in the absence of interactions between the factors, $(x_1y_jz_k) - (x_0y_jz_k)$ has the same expectation for all pairs (j, k). The effective replication of the comparisons between x_0 and x_1 is 12. Even if there are interactions between the factors, the factorial design is still superior to the other two designs, which do not consider that the size of the $x_1 - x_0$ effect depends on the particular combination of Y and Z levels. The results of the first two types of experiment may not be reproducible if the levels of the other two factors are changed.

Latin Square Designs

For some experiments, there may be more than one appropriate blocking scheme that we would like to accommodate in the experiment. One simple and well-known example is the problem of assessing the wear performance of automobile tires. Different brands of tire will perform differently. In addition, tires may be fitted to any one of four positions; there may be differences in performance between the four positions. Finally, there will also be overall differences in performances between different cars. In this situation, we would like to design a single experiment in which we allocate tires to each position for each car so that each brand of tire is tested in each position of each car. This is accomplished with the type of design shown in Fig. 1; this particular design evaluates four brands of tire in four positions for four different cars.

Tire Position	Car I	II	III	IV
Left, Front	A	B	C	D
Right, Front	B	D	A	C
Left, Rear	C	A	D	B
Right, Rear	D	C	B	A

Figure 1. A Latin square design for an experiment to assess the wear of four brands of car tire. A row represents possible positions of a tire on a car, and a column represents one of four varieties of automobile.

The type of experimental design shown in Fig. 1 is referred to as a Latin square design. Latin square designs have the experimental units arranged in a double-blocking classification system. There are x blocks in each system, with each of x treatments occurring once in each block of each block system. The total number of units in the experiment is x^2. For the example shown in Fig. 1, the two blocking systems are the tire position and the type of car, while the brand of tire is the treatment. The two blocking systems in this type of design are traditionally referred to as rows and columns.

In a Latin square design, there are three sources of variability: variability between rows, variability between columns, and variability between treatments. The model for the yield of this type of experiment is

$$y_{jk} = \mu + r_j + c_k + t_{l(j,k)} + \epsilon_{jk}$$

where r_j represents treatment effects within row j, c_k represents treatment effects within column k, and ϵ_{jk} is an error term specifying the deviation of the unit in the row j, column k from the overall average yield μ. Note that the yields y_{jk} are characterized by only two of the three classifications (rows, columns, and treatments). In this type of design, only two classifications are required to uniquely classify each observation; the type of treatment is completely determined by the row and column indices j and k. This dependence is shown in the form of the treatment index $l(j, k)$. The three types of treatment effects may be estimated orthogonally, using the restrictions that

$$\sum_j r_j = 0, \quad \sum_k c_k = 0, \quad \text{and} \quad \sum_l t_l = 0$$

The Latin square design is a solution to the problem of including two blocking factors within a single experiment. However, it is extremely restrictive. The number of replicates of each treatment must be equal to the number of treatments. Furthermore, the number of degrees of freedom for error in the analysis of variance for an experiment with t treatments are $(t - 1)(t - 2)$—this provides only two degrees of freedom for an experiment with three treatments, and six degrees of freedom when there are four treatments. We would not expect to obtain an adequate estimate of σ^2 under these circumstances. When using a Latin square design for an experiment with three or four treatments, it is usually necessary to have more than one square.

Tire Position	Car							
	I	II	III	IV	I	II	III	IV
Left, Front	A	B	C	D	A	B	C	D
Right, Front	B	D	A	C	B	C	D	A
Left, Rear	C	A	D	B	C	D	A	B
Right, Rear	D	C	B	A	D	A	B	C

Figure 2. Variation on a Latin square design to assess the wear of car tires. This design uses multiple Latin squares with common row effects. In this case, the effects of tire position are consistent across both groups of automobile type.

Experiments in which multiple Latin squares are used fall into one of two categories:

1. *Experiments in Which One of the Blocking Systems is Consistent Over Different Squares.* For instance, if two groups of cars are used in the experiment to determine the wear of tires and the four possible tire positions are the second blocking system for each group, we would expect that differences between position should be consistent over the two groups. An experimental design for this situation is shown in Fig. 2.
2. *Experiments in Which the Rows or Columns Have No Relationship to Each Other.* This type of design is shown in Fig. 3.

An index of designs for a given number of treatments and for a set of experimental units with two blocking systems may be found in Ref. 4; detailed treatments of experimental design may be found in Refs. 5 and 6.

APPLICABILITY OF EXPERIMENTAL DESIGN TO SOFTWARE TESTING

Design of experiments can be used in testing software to generate a set of test cases that will produce maximum coverage with respect to the desired criterion (e.g., branch coverage, *c*-uses, *p*-uses). The idea is that as coverage increases, the number of faults found and removed during a test increases, leaving fewer residual faults in the system. Malaiya et al. have studied relationships between test coverage and fault coverage (7). They found the relationships sufficiently well-defined to develop a logarithmic model relating fault coverage to test coverage. According to this model, as test coverage increases, fault coverage also increases. Details of the model are given (7).

Row	Column							
	1	2	3	4	5	6	7	8
1	A	B	C	D				
2	B	D	A	C				
3	C	A	D	B				
4	D	C	B	A				
5					A	B	C	D
6					B	A	D	C
7					C	D	A	B
8					D	C	B	A

Figure 3. Example of a multiple Latin square experiment in which there are no common row or column effects.

One particular method of using experimental design techniques (8) uses combinatorial designs to generate tests that efficiently cover n-way combinations of a system's test parameters (the parameters that determine the system's test scenarios). Cohen et al. show (8) that the number of test cases grows logarithmically in the number of test parameters. This makes it fairly inexpensive to add detail to the test cases in the form of additional parameters. The greedy algorithm for producing test cases given by Cohen et al. (8) is quite straightforward. Suppose that we have a system with k test parameters, and that the ith parameter has l_i distinct values. At this point, we've already selected r test cases, and we wish to select the $(r + 1)^{st}$. This is done by generating M different candidate test cases and then choosing the one that covers the most new n-tuples (e.g., pairs, triples) of parameters. Cohen et al. report (8) that when M was set to 50 (50 candidate test cases were generated for each new test case), the number of generated test cases grew logarithmically in the number of parameters when all the parameters had the same number of values. Furthermore, increasing M did not drastically reduce the number of test cases.

EXPERIMENTAL ASSESSMENT OF SOFTWARE RELIABILITY

To determine a software system's reliability we must conduct three distinct experiments. First, we must know how the software is used; that is, we will need to know the way in which users exercise the operations implemented in the software system. Secondly, we must understand how the system is designed. Finally, we must observe how likely it is for each program module in the system to fail. These issues are explored in more detail in the following sections.

SOFTWARE RELIABILITY: AN OVERVIEW OF THE PROBLEM

Computer programs do not break. They do not fail monolithically. Programs are designed to perform a set of mutually exclusive tasks or functions. Some of these functions work quite well, whereas others may not work well at all. When a program is executing a particular function, it executes a well-defined subset of its code. Some of these subsets are flawed and some are not. Users tend to execute subsets of the total program functionality. Two users of the same software may have totally different perceptions as to the reliability of the same system. One user may use the system on a daily basis and never experience a problem. Another user may have continual problems in trying to execute the same program while attempting to perform different functions.

The literature in reliability is rife with efforts to port hardware reliability notions to software. It just won't work. Software is very different from hardware. Software systems are composed of individual and largely independent sections called modules. At any instant in the life of a program, only one of these modules is capable of demanding the resources of the computer central processing unit (CPU). It would be very difficult to conceive of an analogous hardware system. Imagine, if you will, an automobile capable of moving one piston at a time in its engine, turning but one wheel as it goes down the road, or switching its operation from the distributor to the rear differential to the left tail light, and so on. Software systems are constructed of many modules. Only some of these modules will execute when the software performs its nominal functionality. A module may be hopelessly flawed, but if it never enters the set of operating modules, these flaws will never be seen nor expressed.

Yet another problem we have in the software quality business, we can only have at most one software system. For example, if we build a million cars all with the same program running the ignition controls, each of these cars will have exactly the same program. There is exactly no variation in the manufacture of software. Whatever we do, we will only build one system. There may be zillions of copies, but they are all the same program. This is good and this is bad. If there is one design defect, then each of the zillion copies has this flaw. It is good in that there is exactly zero variation in the manufacturing process.

FOUNDATIONS

One of the most offensive and misleading terms used in the software profession is the notion of a bug. There is exactly no information in the statement that a program has *n* bugs in it. No one have ever defined just what a bug is. A recent check of the National Institute of Standards and Technology revealed no temperature-controlled rubidium standard bug. The whole of software reliability engineering must center around a precise understanding of software flaws and their etiology. Some terminology will be in order.

Errors

An error is an act of a person. A software requirement analyst may write an incorrect specification. The act of introducing the defective specification is the error. A software designer, may fail to implement a specification correctly. The act of omission is the error. There are two principal categories of these errors. There are sins of commission. A person actively introduced a problem. There is also the sin of omission. A person failed to perform some activity that was prescribed.

Faults

People make errors. The physical and tangible result of these errors is a fault. Unfortunately there is no particular definition of precisely what a software fault is. In the face of this difficulty, it is rather hard to develop meaningful associative models between faults and metrics. In calibrating our model, we would like to know how to count faults in an accurate and repeatable manner. In measuring the evolution of the system to talk about rates of fault introduction and removal, we measure in units describing how the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. Since the measurements of system structure are collected at the module level (by module we mean procedures and functions), we would like information about faults at the same granularity. We would also like to know if there are quantities that are related to fault counts that can be used to make our calibration task easier.

Simply put, a fault is a *structural defect* in a software system that may lead to the system's eventually failing. In other words, it is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. More details are given by Nikora (9). Faults are introduced into a system by people making errors in their tasks. Ultimately, if we wish to improve the quality of our software systems we must come to grips with the fact that faults are introduced by people due to the psychological complexity of the specification, design, or coding a piece of software.

A significant amount of work remains to be done in the actual measurement of software faults. To count faults, we needed to develop a method of identification, a standard, that is repeatable, consistent, and identifies faults at the same level of granularity as other structural measurements. This type of fault is simple to count, since it occurs only in one module. In identifying and counting faults, we must deal with faults that span only one module as well as those that span several.

Failure Events

A failure occurs when the software system encounters a software fault during the course of execution. There is a problem, however, with the detection of the failure event. Not all such failures will cause the system to stop executing. The system may well continue executing. The failure event may not have important consequences. It may go undetected. A more insidious software failure is one that initiates a chain of events that will bring the system to its knees at some future time. When the failure is finally made manifest, a very long time has elapsed between when the actual failure event occurred and when it had visible consequences.

Thus, a significant problem in the determination of the reliability of a software system is the precise determination of the failure event. If the failure event is largely unobservable, then the notion of the time between observable failures will have a very large (and undetermined) noise component. There is really no way to measure the elapsed time between events that we cannot observe. The failure event, and the circumstances that surround the failure have proven to be most elusive concepts.

Finally, not all faults will lead to failures. Some faults will be located on execution paths that will never be expressed when the software system is executing in a nominal fashion. We are not interested in these faults. They will never lead to failures. They should not be removed from the system. Each time we alter the system we run a risk of introducing new faults. It is most improbable that we would seek to remove a fault from our software that will have no impact on our system only to trade it for a fault that might well cause the system to fail.

Some Thoughts on Time

Computer software exists only in a three-dimensional world. There is no real concept of time as far as software is concerned. Software is not like wine. It will never improve with age. Software is not like gears in a transmission. It will never wear out. Nor will continual use sand its surface smooth. Any faults in a software system at its birth will still be present at its demise. Furthermore, a software system's exposure to time is a highly variable commodity. Some CPU's are very fast, others are very slow in relation to the fast ones. By hardware measurement standards, we could achieve really reliable software by running it on the slowest possible CPU. The notion of time between failure has no particular relevance in software.

It is most inappropriate, then, to think about a software system breaking at some future time based on our observations of its past performance. As we discuss in the next section, software breaks because of what we chose to do with it in the future, not because of some intrinsic characteristic aging in the system. The future reliability of a system, then, depends entirely on the user and how he or she chooses to use the system.

GETTING THE METAPHOR RIGHT

A main concern in software reliability investigations revolves around the failure event itself. Our current view of software reliability is colored by a philosophical approach that began with efforts to model hardware reliability (see Ref. 10 for more details). Inherent in this approach is the notion that it is possible to identify with some precision this failure event and measure the elapsed time to the failure event. For hardware systems this has real meaning. Take, for example, the failure of a light bulb as discussed earlier. A set of light bulbs can be switched on and a very precise timer started for the time that they were turned on. One by one the light bulbs will burn out and we can note the exact time to failure of each of the bulbs. From these failure data, we can then develop a precise estimate for both the mean time to failure for these light bulbs and a good estimate of the variance of the time to failure. The case for software systems is not at all the same. Failure events are sometimes quite visible in terms of catastrophic collapses of a system. More often than not, the actual failure event will have occurred a considerable time before its effect is noted. In most cases it is simply not possible to determine with any certainty just when the actual failure occurred on a real time clock. The most simple example of this improbability of measuring the time between failures of a program may be found in a program that hangs in an infinite loop. Technically the failure event happened on entry to the loop. The program, however, continues to execute until it is killed. This may take seconds, minutes, or hours depending on the patience and/or attentiveness of the operator. As a result, the accuracy of the actual measurement of time intervals is a subject never mentioned in most software validation studies [details given by Chan, Littlewood, Brocklehurst, and Snell (11)]. The bottom line for the measurement of time between failures in software systems is that we cannot measure with any reasonable degree of accuracy these time intervals. This being the case, we then must look to new metaphors for software systems that will permit us to model the reliability of these systems based on things that we *can* measure with some accuracy.

Yet another problem with the hardware adaptive approach to software reliability modeling is that the failure of a computer software system is simply not time dependent. A system can operate without failure for years and then suddenly become very unreliable based on the changing functions that the system must execute. Many university computer centers experienced this phenomenon in the late 1960s and early 1970s when there was a sudden shift in computer science curricula from programming languages such as FORTRAN that had static run time environments to ALGOL derivatives such as Pascal and Modula that had dynamic run time environments. From an operating system perspective, there was a major shift in the functionality of the operating system exercised by these two different environments. As the shift was made to the ALGOL-like languages, latent code in the operating system, specifically those routines that dealt with memory management, that had not been executed overly much in the past now became central to the new operating environment. This code was both fragile and untested. The operating systems that had been so reliable began to fail like cheap light bulbs.

A new metaphor for software systems would focus on the functionality that the code is executing and not the software as a monolithic system. In computer software systems, it is the functionality that fails. Some functions may be virtually failure free, whereas other functions will collapse with certainty whenever they are executed. It is possible to measure the activities of a system as it executes its various functions and characterize the reliability of the system in terms of these functionalities.

Each program functionality may be thought of as having an associated reliability estimate. We may chose to think of the reliability of a system in these functional terms. Users of the software system, however, have a very different view of the system. What is important to the user is not that a particular function is fragile or reliable, but rather whether the system will operate to perform those actions that the user will want the system to perform correctly. From a user's perspective, it matters not, then, that certain functions are very unreliable. It only matters that the functions associated with the user's actions or operations are reliable. The classical example of this idea was expressed by the authors of the early UNIX utility programs. In the last paragraph of the documentation for each of these utilities was a list of known bugs for that program. In general, these bugs were not a problem. Most involved aspects of functionality that the typical user would never exploit.

From a functional viewpoint, a program may be viewed as a set of program modules that are executing a set of mutually exclusive functions. If the program executes a functionality consisting of a subset of these modules that are fault free, it will never fail no matter how long it executes this functionality. If, on the other hand, the program is executing a functionality that contains fault laden modules, there is a very good likelihood that it will fail whenever that functionality is expressed [details given by Munson (12)]. Furthermore, it will fail with certainty when the right aspects of functionality are expressed.

The main problem in the understanding of software reliability from this new perspective is getting the granularity of the observation right. Software systems are designed to implement each of their functionalities in one or more code modules. In some cases there is a direct correspondence between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceability of functionality to modules. The functionality will be expressed in many different code modules. It is the individual code module that fails. A code module will, of course, be executing a particular functionality when it fails. We must come to understand that it is the functionality that fails.

As a program is exercising any one of its many functionalities in the normal course of operation of the program, it will apportion its time across this set of functionalities (see Ref. 12 for more detail). The proportion of time that a program

Table 5. Example of the IMPLEMENTS Relation

$O \times F$	f_1	f_2	f_3	f_4
o_1	T	T		
o_2		T	T	T

spends in each of its functionalities is the *functional profile* of the program. Furthermore, within the functionality, it will apportion its activities across one to many program modules. This distribution of processing activity is represented by the concept of the execution profile. In other words, if we have a program structured into n distinct modules, the *execution profile* for a given functionality will be the proportion of program activity for each program module while the function was being expressed.

As the discussion herein unfolds, we see that the key to understanding program failure events is the direct association of these failures to execution events with a given functionality. A Markovian stochastic process will be used to describe the transition of program modules from one to another as a program expresses a functionality. From these observations, it will become fairly obvious just what data will be needed to describe accurately the reliability of the system. In essence, the system will be able to appraise us of its own health. The reliability modeling process is no longer something that will be performed *ex post facto*. It may be accomplished dynamically while the program is executing.

Operations

To assist in the subsequent discussion of program functionality, it will be useful to make this description somewhat more precise by introducing some notation conveniences. Assume that the software system S was designed to implement a specific set of mutually exclusive functionalities F. Thus, if the system is executing a function $f \in F$, then it cannot be expressing elements of any other functionality in F. Each of these functions in F was designed to implement a set of software specifications based on a user's requirements. From a user's perspective, this software system will implement a specific set of operations, O. This mapping from the set of user perceived operations, O, to a set of specific program functionalities, F, is one of the major tasks in the software specification process.

Each operation that a system may perform for a user may be thought of as having been implemented in a set of functional specifications. There may be a one-to-one mapping between the user's notion of an operation and a program function. In most cases, however, there may be several discrete functions that must be executed to express the user's concept of an operation. For each operation, o, that the system may perform, the range of functionalities, f, must be well known. Within each operation one or more of the system's functionalities will be expressed. For a given operation, o, these expressed functionalities are those with the property

$$F^{(o)} = \{f : F \mid \forall \text{IMPLEMENTS}(o, f)\}$$

It is possible, then, to define a relation IMPLEMENTS over $O \times F$ such that IMPLEMENTS(o, f) is true if functionality f is used in the specification of an operation, o. For each operation $o \in O$, there is a relation p' over $O \times F$ such that $p'(o, f)$ is the proportion of activity assigned to functionality f by operation o. An example of the IMPLEMENTS relation for two operations implemented in four specified functions is shown in Table 5. In this table, we can see that functions f_1 and f_2 are used to implement the operation o_1.

In Table 6, there is an example of the relation p'. These numbers represent the proportion of time each of the functions will execute under each of the operations. The software design process is a matter of assigning functionalities in F to specific program modules $m \in M$, the set of program modules. The design process may be thought of as the process of defining a set of relations, ASSIGNS over $F \times M$ such that ASSIGNS(f, m) is true if functionality f is expressed in module m. For a given software system, S, let M denote the set of all program modules for that system. For each function $f \in F$, there is a relation p over $F \times M$ such that $p(f, m)$ is the proportion of execution events of module m when the system is executing function f. Table 7 shows an example of the ASSIGNS relation for the four functions presented in Table 5. In this example we can see the function f_1 has been implemented in the program modules m_1, m_2 and m_4. One of these modules, m_1, will be invoked regardless of the functionality. It is common to all functions. Other program modules, such as m_2, are distinctly associated with a single function. In Table 8, there is an example of the relation p. These numbers represent the proportion of time each of the functions will execute in each of the program modules. The row marginal values represent the total proportion of time allocated to each of the functions. These are the same values as the column marginals of Table 6. Similarly, the column marginal values of Table 8 represent the proportion of time distributed across each of the six program modules.

There is a relationship between program functionalities and the software modules that they will cause to be executed. These program modules will be assigned to one of three distinct sets of modules that, in turn, are subsets of M. Some

Table 6. Example of the p' Relation

$p'(o, f)$	f_1	f_2	f_3	f_4
o_1	0.2	0.8	0	0
o_2	0	0.4	0.4	0.2

Table 7. Example of the ASSIGNS Relation

$F \times M$	m_1	m_2	m_3	m_4	m_5	m_6
f_1	T	T		T		
f_2	T		T		T	
f_3	T		T			T
f_4	T		T		T	T

Table 8. Example of the p Relation

$p(f, m)$	m_1	m_2	m_3	m_4	m_5	m_6
f_1	1	1	0	1	0	0
f_2	1	0	1	0	0.1	0
f_3	1	0	0.5	0	0	0.3
f_4	1	0	1	0	0.4	0.1

modules may execute under all of the functionalities of S. This will be the set of common modules. The main program is an example of such a module that is common to all operations of the software system. Essentially, program modules will be members of one of two mutually exclusive sets. There is the set of program modules M_c of common modules and the set of modules M_F that are invoked only in response to the execution of a particular function. The set of common modules, $M_c \subset M$ is defined as those modules that have the property

$$M_c = \{m : M | \forall f \in F \bullet \text{ASSIGNS(f, m)}\}$$

All of these modules will execute regardless of the specific functionality being executed by the software system. Yet another set of software modules may or may not execute when the system is running a particular function. These modules are said to be potentially involved modules. The set of potentially involved modules is

$$M_p^{(f)} = \{ \quad m : M_F | \exists f \in F \bullet \text{ASSIGNS}(f, m) \wedge 0 < p(f, m) < 1\}$$

In other program modules, there is extremely tight binding between a particular functionality and a set of program modules. That is, every time a particular function, f, is executed, a distinct set of software modules will always be invoked. These modules are said to be indispensably involved with the functionality f. This set of indispensably involved modules for a particular functionality, f, is the set of those modules that have the property that

$$M_i^{(f)} = \{m : M_F | \forall f \in F \bullet \text{ASSIGNS}(f, m) \Rightarrow p(f, m) = 1\}$$

As a direct result of the design of the program, there will be a well defined set of program modules, M_f, that might be used to express all aspects of a given functionality, f. These are the modules that have the property that

$$m \in M_f = M_c \cup M_p^{(f)} \cup M_i^{(f)}$$

From the standpoint of software design, the real problems in understanding the dynamic behavior of a system are not necessarily attributable to the set of modules, M_i, that are tightly bound to a functionality or to the set of common modules, M_c, that will be invoked for all executing processes. The real problem is the set of potentially invoked modules, M_p. The greater the cardinality of this set of modules, the less certain we may be about the behavior of a system performing that function. For any one instance of execution of this functionality, a varying number of the modules in M_p may execute.

Profiles of Software Dynamics

When a program begins the execution of a functionality, we may envision this beginning as the start of a stochastic process. It is possible to construct a probability adjacency matrix, P, whose entries represent the transition probability from each module to another module at each epoch in the execution process while a particular functionality is executing.

The transition from one module to another may be described as a stochastic process. In which case we may define an indexed collection of random variables $\{X_t\}$, where the index t runs through a set of nonnegative integers, $t = 0, 1, 2, \ldots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly one of its M modules. The fact of the execution occurring in a particular module is a *state* of the system. For this software system, the system is found in exactly one of a finite number of mutually exclusive and exhaustive states that may be labeled $0, 1, 2, \ldots, M$. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0, 1, 2, \ldots$ and where each random variable may take on any one of the $(M + 1)$ integers, from the state space $A = \{0, 1, 2, \ldots, M\}$.

A stochastic process $\{X_t\}$ is a Markov chain if it has the property that

$$\Pr[X_{t+1} = j | X_t = i_t, X_{t-1} = i_{t-1}, \cdots, X_0 = i_0] \\ = \Pr[X_{t+1} = j | X_t = i_t]$$

for any epoch $t = 0, 1, 2, \ldots$ and all states $i_0, i_1, \ldots, i_t$ in the state space A. This is equivalent to saying that the conditional probability of executing any module at any future epoch is dependent only on the current state of the system. The conditional probabilities $\Pr[X_{t+1} = j | X_t = i_t]$ are called the transition probabilities. In that this nomenclature is somewhat cumbersome, let $p_{ij}^{(n)} = \Pr[X_n = j | X_{n-1} = i]$. Within the execution of a given functionality, the behavior of the system is static. That is, the transition probabilities do not change from one epoch to another. Thus,

$$\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$$

for i, j, in S, which is an additional condition of a Markov process.

Since the $p_{ij}^{(n)}$ are conditional probabilities it is clear that

$$p_{ij}^{(n)} \geq 0, \quad \text{for all } i, j \text{ in } A, \quad n = 0, 1, 2, \cdots$$

and,

$$\sum_{j=0}^{M} p_{ij}^{(n)} = 1, \quad \text{for all } i \text{ in } A \text{ and } n = 0, 1, 2, \cdots$$

Interestingly enough, for all software systems there is a distinguished module, the main program module that will always receive execution control from the operating system. If we denote this main program as module 0 then,

$$\Pr[X_0 = 0] = 1 \quad \text{and} \quad \Pr[X_0 = i] = 0 \quad \text{for } i = 1, 2, \cdots, M$$

We can see, then, that the unconditional probability of executing in a particular module j is

$$\Pr[X_n = j] = p_{ij}^{(n)} \Pr[X_0 = 0] = p_{ij}^{(n)}$$

The problem of the determination of the transition probabilities

$$p_{ij}^{(0)} = \Pr[X_1 = j | X_0 = i]$$

of $\mathbf{P}^0$ is now of interest. Each row i of $\boldsymbol{P}$ represents the probability of the transition to a new state j given that the program is currently in state i. These are mutually exclusive events. The program may only transfer control to exactly one other program module. Under this assumption, the conditional probabilities that are the rows of $\mathbf{P}^0$, also have the property that they are distributed multinomially. They profile the transitions from one state to another.

The granularity of the term, epoch, is an important consideration. An epoch begins with the onset of execution in a particular module and ends when control is passed to another module. The measurable event for modeling purposes is this transition among the program modules. We will count the number of calls from a module and the number of returns to that module. Each of these transitions to a different program module from the one currently executing will represent an incremental change in the epoch number. Computer programs executing in their normal mode will make state transitions between program modules rather rapidly. In terms of real clock time, many epochs may elapse in a relatively short period.

Operational Profiles

Any software system has at its core a set of operations O that it was designed to implement. Each user will typically exercise a subset of these functionalities. Each user will probably use each operation to a different extent than every other user. The users bring to the system an operational profile of his/her use of the system.

The operation profile of the software system is the set of unconditional probabilities of each of the functionalities O being executed by the user. Let W be a random variable defined on the indices of the set of elements of O. Then, $p_k = \Pr[W = m]$, $m = 1, 2, \ldots, \|O\|$ is the probability that the user is executing program operation m as specified in the functional requirements of the program and $\|O\|$ is the cardinality of the set of operations.

Functional Profiles

When a software system is constructed by the software developer, it is designed to fulfill a set of specific functional requirements. The user will run the software to perform a set of perceived operations. Each of the operations, o, maps to one or more elements in the set of functionalities as defined by the IMPLEMENTS relation. The functional profile of the software system is the set of unconditional probabilities of each of the functionalities F being executed by the user under that user's operational profile. Let Y be a random variable defined on the indices of the set of elements of F. Then, $q_k = \Pr[Y = k]$, $k = 1, 2, \ldots, \|F\|$ is the probability that the user is executing program functionality k as specified in the functional requirements of the program and $\|F\|$ is the cardinality of the set of functions [described by Musa (13)]. A program executing on a serial machine can only be executing one functionality at a time. The distribution of q, then, is multinomial for programs designed to fulfill more than two specific functions. The prior knowledge of this distribution of functions should guide the software design process [details given by Munson and Ravenel (14)].

Execution Profiles

When a program is executing a given functionality, say f_k, it will distribute its activity across the set of modules, M_{f_k}. At any arbitrary epoch, n, the program will be executing a module $m_i \in M_{f_k}$ with a probability, $u_{ik} = \Pr[X_n = i|Y = k]$. The set of conditional probabilities $u_{\bullet k}$ where $k = 1, 2, \ldots, \#\{F\}$ constitute the execution profile for function f_k. As was the case with the functional profile, the distribution of the execution profile is also multinomial for a software system consisting of more than two modules. As a matter of the design of a program, there may be a nonempty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised. Of course, this will cause the cardinality of the set M_f to vary. A particular execution may not invoke any of the modules of $M_p^{(f)}$. On the other hand, all of the modules may participate in the execution of that functionality. This variation in the cardinality of M_f within the execution of a single functionality will contribute significantly to the amount of test effort that will be necessary to test such a functionality.

Each operation will be implemented by a subset of functionalities, i.e., $F_e^{(o)} \subset F$. As each operation is run to completion it will generate an execution profile. This execution profile may represent the results of the execution of one or more functions. Most operations, however, do not exercise precisely one functionality. Rather, they may apportion time across a number of functionalities. For a given operation, let l be a proportionality constant. Then, $0 \le l_k \le 1$ will represent the proportion of epochs that will be spent executing the kth functionality in $F_e^{(o)}$. Thus an operational profile of a set of modules will represent a linear combination of the conditional probabilities, u_{ik} as follows:

$$p_i = \sum_{f_k \in F_e^{(o)}} l_k u_{ik}$$

Module Profiles

The manner in which a program will exercise its many modules as the user chooses to execute the functionalities of the program is determined directly by the design of the program. Indeed, this mapping of functionality onto program modules is the overall objective of the design process. The *module profile, s*, is the unconditional probability that a particular module will be executed based on the design of the program. It is derived through the application of Bayes' rule. First, the joint probability that a given module is executing and the program is exercising a particular function is given by

$$\Pr[X_n = j \cap Y = k] = \Pr[Y = k]\Pr[X_n = j|Y = k] = q_k u_{ik}$$

where j and k are defined as before. Thus, the unconditional probability, s_i, of executing module j under a particular design is

$$\begin{aligned} s_i &= \Pr[X_n = i] \\ &= \sum_k \Pr[X_n = i \cap Y = k] \\ &= \sum_k q_k u_{ik} \end{aligned}$$

As was the case for the functional profile and the execution profile, only one module can be executing at any one time. Hence, the distribution of q is also multinomial for a system consisting of more than two modules.

Failure Profiles

What is needed now is a mechanism to describe the actual failure event in a software system. As was noted earlier, a failure will actually occur through the execution of a fault in a program module. A reasonable and viable mechanism for us to use to capture the fault event is to imagine the existence of an hypothetical failure module. A failure event in any module may then be represented as a transition to this absorbing failure state in our Markov process model for the program operation. With this concept we will augment our module transition matrix $\mathbf{P}$ to form the new augmented matrix $\mathbf{P}'$ containing a new row and column for the module representing the failure state.

Each failure of the program will alter our view of the transition probabilities of a particular module to the failure state module.

THE CONDUCT OF INQUIRY

Failure Measurement

Each and every failure event must be assiduously monitored and recorded. As was indicated earlier, the vast majority of software failures will never be observed nor recorded. The logical mechanism for trapping and recording faults at their point of origin is provided by the exception handling facility such as that offered by Ada (15). Using this mechanism, we may instrument each module for the possible failure conditions and record these failures at the point of origin. The alternative to this dynamic measurement opportunity is indeed distressing. If the software is not appropriately instrumented for failure recording, each failure must be traced to its module of origin by hand. This is a most laborious and error prone activity. In either event, we must record each failure of the system and ascribe this failure to a module.

Measurement of Profiles

Let us now turn to the measurement scenario for the modeling process described above. Consider a system whose requirements specify a set of a user operations. These operations, again specified by a set of functional requirements, will be mapped into a set of b elementary program functions. The functions, in turn, will be mapped by the design process into a set of m program modules.

The software is designed to function optimally under an a priori operational profile. We need a mechanism for tracking the actual behavior of the user of the system. To this end we require a vector, $\boldsymbol{O}$, in which the program will count the frequency of each of the operations. That is, an element o_i of this vector will be incremented every time the program initiates the ith user operation. Each of the operations is distinct and they are mutually exclusive. Thus we may use the Bayesian estimation process to compute estimates for the actual posterior operational profile for the software.

The next, static, matrix $\boldsymbol{Q}$ that will have to be maintained is the matrix that describes the mapping $O \times F$ of the operations to the program functions. Each element of this matrix will have the property that

$$q_{ij} = \begin{cases} 1 & \text{if IMPLEMENTS}\,(o_i,\ f_j)\ \text{is TRUE} \\ 0 & \text{if IMPLEMENTS}\,(o_i,\ f_j)\ \text{is FALSE} \end{cases}$$

The next, static, matrix $\boldsymbol{S}$ that will have to be maintained is the matrix that describes the mapping $F \times M$ of the function to the program modules as a result of the program design process. Each element of this matrix will have the property that

$$s_{jk} = \begin{cases} 1 & \text{if ASSIGNS}\,(f_j,\ m_k)\ \text{is TRUE} \\ 0 & \text{if ASSIGNS}\,(f_j,\ m_k)\ \text{is FALSE} \end{cases}$$

A current assessment of the frequency with which functions are executed may be maintained in a matrix $\boldsymbol{S}$. As was the case with the operational profile, an element o_j of this vector will be incremented every time the program initiates the jth function.

Finally, we need to record the behavior of the total system as it transitions from one program module to another. If there are a total of m modules, then we will need an $n \times n$ ($n = m + 1$) matrix $\boldsymbol{T}$ to record these transitions. Whenever the program transfers control from module m_i to module m_j the element t_{ij} of $\boldsymbol{T}$ will increase by one. The rows of the transition matrix $\boldsymbol{P}$ may be obtained dynamically from the estimation methods presented above.

The index pair, (i, j), constitutes the transition event from module m_i to module m_j. If we preserve the sequence of (i, j) from the 0th epoch to the present, we will be able to reconstruct the functional (operational) sequences of program behavior. However, in that the number of epochs will be extremely large for reliable program operation, an alternate mechanism might be to preserve simply the last n (i, j) pairs. This may be done by pushing each of the (i, j) pairs onto a stack, $\boldsymbol{C}$, of length n that will preserve only the (i, j) pairs of the last n epochs.

Thus, the essential measurements components will consist of the components, $\boldsymbol{C}$, $\boldsymbol{O}$, $\boldsymbol{Q}$, $\boldsymbol{S}$, and $\boldsymbol{T}$. From these data elements we may construct the functional behavior of any system through its final n epochs. If a program is augmented to include these fundamental matrices and mechanism, either within the operating system or even the program itself, to record the necessary entries in these matrices, it would be possible to reconstruct the function that a program was executing when it met its untimely demise.

Instrumentation for Measurement

Software reliability is a dynamic consideration. A source code program has never been known to fail. It will fail only when it is compiled and executed. Therefore, we must be equipped to measure the program when it is running. There are two types of tools that will assist in this measurement. We may instrument the software with probes or we may measure the behavior of the system through the use of hardware measurement tools, probes.

Software Probes. To measure the activity within a program we must insert special call statements at selected places in the software depending on what we wish to measure. These calls are the software probes. Each call will cause control to be switched to a tally function that will record the call event.

The actual recording that occurs in the object functions of the software probes is dependent on the nature of the event we wish to monitor. For software reliability purposes, we are interested in instrumenting the software to record functionality information and also module transition information. To instrument the software for functional profile information, the user must physically determine the beginning of the set of modules in call tree representing each functionality. In this case, calls to the tally function will record the frequency that each functionality has executed. In the case of the execution profiles, These call statements transfer control to a special function that records the entry event to each module in a frequency transition matrix.

It takes two levels of software to handle the software probes. First, there is a preprocessor that physically inserts the necessary calls into the source code. Second, there is the runtime support consisting of the instrumentation package for cumulating the software transition information. This runtime package will also typically impose some input/output burden on the system, as well, in that we need to periodically dump the transition matrix in that the system may fail at any time taking the recording module with it.

Software probes have a very definite problem. They have the disadvantage of being obtrusive. The system will take a real performance hit with theses probes in place. This is particularly true when we are instrumenting a poorly designed system that employs a number of modules that a called very frequently.

Hardware Probes. An alternative method for monitoring the activity of an executing program is to obtain the necessary call information directly from the instruction stream between the CPU and main memory. Each call is typically initiated by a distinct P-capturing instruction. These instructions and their call addresses may be obtained directly from the flow of instructions between memory and the CPU. Certainly, the overwhelming advantage of this approach is that it is unobtrusive. It does not impact the performance of the software it is monitoring. The downside, is that the hardware costs are substantial. We must purchase both a hardware probe and a separate machine to process the flow of information from the probe.

EXPERIMENTAL OBJECTIVES

If we wish to understand the reliability of a software system and make future predictions about its behavior we must conduct three distinct experiments designed to reveal how the software will be used, how was it designed, and how likely each program module is to fail when it is executed. The success of these experiments will be determined largely by our ability to obtain accurate measurements on the program and its users. We must learn that the accuracy of our reliability assessments depend entirely on the accuracy of our measurements.

Understanding Software Behavior

To develop a viable assessment of the reliability of a software system we must conduct three distinct experiments. First, we must know how the software will be used. More precisely, we will need good estimates for the operational profile for users and the variance of these profiles across all users. In the best case everyone will use the software in exactly the same manner. In the worst possible case, each user will exercise a different set of operations. It should be quite clear to us by now that the reliability of the system will be determined, in the main, by the ability of the system to operate correctly for each user. This first experiment will yield accurate operational profile information.

The second experiment will be to determine the behavior of the system under the observed operational profiles. During this phase, we will execute the system and measure its behavior to learn about the way in which it was designed. We would like to understand the nature of the distribution of the module profiles. These module profiles, of course, are dependent on functional profiles and execution profile. Each operational profile will cause the system to exercise a particular set of functionalities. Each of these functionalities, in turn, will generate an execution profile.

The third experiment will focus on obtaining reasonable estimates for the module failure profiles. These failure profiles represent the probability of transitioning to the virtual failure modules from any of the program modules. To do this, we must very carefully map each observed failure to a particular module.

Point Estimates for Profiles

The focus will now shift to the problem of understanding the nature of the distribution of the probabilities for various profiles. We have so far come to recognize these profiles in terms of their multinomial nature. The multinomial distribution is useful for representing the outcome of an experiment involving a set of mutually exclusive events. Let $S = \bigcup_{i=1}^{M} S_i$ where S_i is one of M mutually exclusive sets of events. Each of these events would correspond to a program executing a particular module in the total set of program modules. Further, let $\Pr(S_i) = w_i$ and

$$w_T = 1 - w_1 - w_2 - \cdots - w_M$$

under the condition that $T = M + 1$, as defined earlier. In which case w_i is the probability that the outcome of a random experiment is an element of the set S_i. If this experiment is conducted over a period of n trials then the random variable X_i will represent the frequency of S_i outcomes. In this case, the value, n, represents the number of transitions from one program module to the next. Note that

$$X_T = n - X_1 - X_2 - \cdots - X_M$$

This particular distribution will be useful in the modeling of a program with a set of k modules. During a set of n program steps, each of the modules may be executed. These, of course, are mutually exclusive events. If module i is executing then module j cannot be executing.

The multinomial distribution function with parameters n and $\mathbf{w} = (w_1, w_2, \ldots, w_T)$ is given by

$$f(\mathbf{x}|n, \mathbf{w}) = \begin{cases} \dfrac{n!}{\prod_{i=1}^{k-1} x_i!} w_1^{x_1} w_2^{x_2} \cdots w_M^{x_M}, \\ \qquad\qquad (x_1, x_2, \cdots, x_M) \in S \\ 0 \quad \text{elsewhere} \end{cases}$$

where x_i represents the frequency of execution of the ith program module.

The expected values for the x_i are given by

$$E(x_i) = \bar{x}_i = nw_i,\ i = 1, 2, \ldots, k$$

the variances by

$$\mathrm{var}(x_i) = nw_i(1 - w_i)$$

and the covariance by

$$\mathrm{cov}(w_i, w_j) = -nw_iw_j, \quad i \neq j$$

We would like to come to understand, for example, the multinomial distribution of a program's execution profile while it is executing a particular functionality. The problem, here, is that every time a program is run we will observe that there is some variation in the profile from one execution sample to the next. It will be difficult to estimate the parameters $\mathbf{w} = (w_1, w_2, \ldots, w_T)$ for the multinomial distribution of the execution profile. Rather than estimating these parameters statically, it would be far more useful to us to get estimates of these parameters dynamically as the program is actually in operation, hence the utility of the Bayesian approach.

To aid in the process of characterizing the nature of the true underlying multinomial distribution, let us observe that the family of Dirichlet distributions is a conjugate family for observations that have a multinomial distribution [details in Wilks (16)]. The probability distribution function (pdf) for a Dirichlet distribution, D(α, α_T), with a parametric vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_M)$ where ($\alpha_i > 0$; $i = 1, 2, \ldots, M$) is

$$f(w|\alpha) = \frac{\Gamma(\alpha_1 + \alpha_2 + \cdots + \alpha_M)}{\prod_{i=1}^{M} \Gamma(\alpha_i)} w_1^{\alpha_1 - 1} w_2^{\alpha_2 - 1} \cdots w_M^{\alpha_M - 1}$$

where ($w_i > 0$; $i = 1, 2, \ldots, M$) and $\sum_{i=1}^{M} w_i = 1$. The expected values of the w_i are given by

$$E(w_i) = \mu_i = \frac{\alpha_i}{\alpha_0} \tag{23}$$

where $\alpha_0 = \sum_{i=1}^{T} \alpha_i$. In this context, α_0 represents the total epochs. The variance of the w_i is given by

$$\mathrm{var}(w_i) = \frac{\alpha_i(\alpha_0 - \alpha_i)}{\alpha_0^2(\alpha_0 + 1)} \tag{24}$$

and the covariance by

$$\mathrm{Cov}(w_i, w_j) = \frac{\alpha_i\alpha_j}{\alpha_0^2(\alpha_0 + 1)}$$

Within the set of expected values μ_i, $i = 1, 2, \ldots, T$, not all of the values are of equal interest. We are interested, in particular, in the value of μ_T. This will represent the probability of a transition to the terminal failure state from a particular program module.

The value of the use of the Dirichlet conjugate family for modeling purposes is twofold. First, it permits us to estimate the probabilities of the module transitions directly from the observed transitions. Secondly, we are able to obtain revised estimates for these probabilities as the observation process progresses. Let us now suppose that we wish to model the behavior of a software system whose execution profile has a multinomial distribution with parameters n and $\mathbf{W} = (w_1, w_2, \ldots, w_M)$ where n is the total number of observed module transitions and the values of the w_i are unknown. Let us assume that the prior distribution of $\boldsymbol{W}$ is a Dirichlet distribution with a parametric vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_M)$ where ($\alpha_i > 0$; $i = 1, 2, \ldots, M$). Then the posterior distribution of $\boldsymbol{W}$ for the behavioral observation $\mathbf{X} = (x_1, x_2, \ldots, x_M)$ is a Dirichlet distribution with parametric vector $\alpha^* = (\alpha_1 + x_1, \alpha_2 + x_2, \ldots, \alpha_M + x_M)$ [details in (17) DeGroot]. As an example, suppose that we now wish to model the behavior of a large software system with such a parametric vector. As the system makes sequential transitions from one module to another, the posterior distribution of $\boldsymbol{W}$ at each transition will be a Dirichlet distribution. Further, for $i = 1, 2, \ldots, T$ the ith component of the augmented parametric vector α will be increased by 1 unit each time module m_i is executed.

BIBLIOGRAPHY

1. E. A. Elsayed, *Reliability Engineering,* Reading, MA: Addison-Wesley, 1996.
2. W. R. Dillon and M. G. Goldstein, *Multivariate Analysis,* New York: Wiley, 1984.
3. R. V. Hogg and A. T. Craig, *Introduction to Mathematical Statistics,* Upper Saddle River, NJ: Prentice-Hall, 1995.
4. W. G. Cochran and G. M. Cox, *Experimental Designs,* New York: Wiley, 1957.
5. G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building,* New York: Wiley, 1978.
6. K. Hinkelmann and O. Kempthorne, *Design and Analysis of Experiments: Introduction to Experimental Design,* New York: Wiley, 1994.
7. Y. K. Malaiya et al., The Relationship Between Test Coverage and Reliability, *Proc. 1994 IEEE Int. Symp. Softw. Reliability Eng.,* Monterey, CA, 1994, pp. 186–195.
8. D. M. Cohen et al., The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Trans. Softw. Eng.,* **23**: 437–444, 1997.
9. A. P. Nikora, Software system defect content prediction from development process and product characteristics, Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, 1998.
10. J. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application,* New York: McGraw-Hill, 1987.
11. P. Y. Chan, B. Littlewood, and J. Snell, Recalibrating Software Reliability Models, *IEEE Trans. Softw. Eng.,* **16**: 458–470, 1990.
12. J. C. Munson, Software Measurement: Problems and Practice, *Ann. Softw. Eng.,* **1**: 255–285, 1995.
13. J. D. Musa, Operational Profiles in Software Reliability Engineering, *IEEE Software,* **10** (2): 14–32, 1993.
14. J. C. Munson and R. H. Ravenel, Designing Reliable Software, *Proc. 1993 IEEE Int. Symp. Softw. Reliability Eng.,* Denver, CO, 1993, pp. 45–54.
15. Reference Manual, *Ada Programming Language,* US Dept. Defense, Washington, D.C., November 1980.
16. S. S. Wilks, *Mathematical Statistics,* New York: Wiley, 1962.
17. M. H. DeGroot, *Optimal Statistical Decisions,* New York: McGraw-Hill, 1970.

ALLEN P. NIKORA
Jet Propulsion Laboratory
JOHN C. MUNSON
University of Idaho

REMOTE AND DISTRIBUTED COMPUTING TOOLS. See REMOTE PROCEDURE CALLS.

REMOTE CONTROL, ROBOTICS. See TELEROBOTICS.

REMOTE NUMERICAL CONTROL MACHINING. See TELECONTROL.