

8.0 RELIABILITY GROWTH AND DEMONSTRATION TESTING

Reliability growth testing is performed to assess current reliability, identify and eliminate faults, and forecast future reliability. The reliability figures are compared with intermediate reliability objectives to measure progress so that resources can be directed to achieve the reliability goals in a timely and cost-effective manner. Whenever a failure occurs, corrective action is undertaken to remove the cause. For hardware, growth testing is the process of testing the equipment under both natural and induced environmental conditions to discover latent failure modes and mechanisms to ensure that all performance, design, and environmental problems have been resolved.

Reliability demonstration is employed toward the end of the growth testing period to verify that a specific reliability level has been achieved. During a demonstration test, the software code is frozen, just as it would be in field use.

Software growth testing and demonstration testing should be performed under the same conditions as field use. That is, the environment in which the software executes must emulate what the software will experience in the field, and environmental conditions must be maintained throughout the test period.

8.1 Software Operational Profile.

The software execution environment includes the hardware platform, the operating system software, the system generation parameters, the workload, and the operational profile. The operational profile is described in detail in Section 9.

Software reliability testing is based on selecting input states from an input space. An input state is a set of input variable values for a particular run. Each input variable has a declared data type (a range and ordering of permissible values). The set of all possible input states for a program is the input space. Each input state is a point in the input space. An operational profile is a function p that associates a probability $p(i)$ with each point i in an input space I . Since the points in the input space are mutually exclusive and exhaustive, all the probabilities must add up to one:

$$\sum_{i \in I} p(i) = 1 \quad (8.1)$$

Example:

To illustrate the operational profile concept, consider a program with three input variables. Each is of data type Boolean, meaning that it has two possible values: TRUE or FALSE. The input space has eight points:

(FALSE,FALSE,FALSE),	(FALSE,FALSE,TRUE),
(FALSE,TRUE,FALSE),	(FALSE,TRUE,TRUE),
(TRUE,FALSE,FALSE),	(TRUE,FALSE,TRUE),
(TRUE,TRUE,FALSE),	(TRUE,TRUE,TRUE).

Letting T stand for TRUE and F for FALSE, an operational profile for the program might look like:

$$p(FFF) = 0.1$$

$$p(FFT) = 0.2$$

$$p(FTF) = 0.1$$

$$p(FTT) = 0.3$$

$$p(TFF) = 0.025$$

$$p(TFT) = 0.2$$

$$p(TTF) = 0.025$$

$$p(TTT) = 0.05$$

The distribution of input states is thus established by the operational profile. This is an *explicit* profile, as described in Section 9.

During growth and demonstration testing the operational profile must be kept stationary (i.e., the $p(i)$'s should not change). The input states chosen for test cases should form a random sample from the input state in accordance with the distribution of input states that the operational profile specifies.

It is generally not practical to fully express or specify an operational profile, because the number of input states for even a simple program can be unworkable. As an example, if a program has three input variables, each of which is a 32-bit integer, the number of distinct input states is

$$2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

Once the operational profile is established, a procedure for selecting a random sample of input states is required, so that test cases can be generated for growth testing and demonstration testing. Random input-state selection is recommended for selecting the input states during testing.

It may be desirable to test several operational profiles that represent the variation in use that can occur among different system installations to determine the resulting variation in reliability.

8.2 Random Input-State Selection.

The operational profile is used to select operations in accordance with their occurrence probabilities. Testing driven by an operational profile is very efficient because it identifies failures, on average, in order of how often they occur. This approach rapidly increases reliability per unit of execution time because the failures that occur most frequently are caused by the faulty operations used most frequently.

Selection should be with replacement for operations and run types by allowing reselection of an element from the population. Because the number of operations is relatively small, at least one of them is likely to be repeated. But the run types will almost certainly be different and the failure behavior may also differ. In general, run categories should be selected with replacement.

Selecting operations. Random selection is feasible for operations with key input variables that are not difficult to change. However, some key input variables can be very difficult and expensive to change, such as one that represents a hardware configuration. In that case some key input variables must be selected deterministically.

Selecting within operations. Consider partitioning the operations into run categories. If there is limited interaction among the input variables with respect to failure behavior, it may be possible to use statistical experimental design techniques to reduce the number of run categories that must be selected. Because the goal is to reduce the number of selections, these should be made without replacement. One experimental design approach uses orthogonal arrays to set up test input states. This approach assumes that failures are influenced only by the variables themselves (A, B, and C) and their pair-wise interactions (AB, AC and BC). Criteria for determining in practice how to select input variables using orthogonal arrays and related techniques is the subject of current research as of 1996.

During growth and demonstration testing, the software must be exercised with inputs randomly selected from a specified operational profile or, if appropriate, from a specified functional profile. The methods described here can be followed for either an operational profile or a functional profile (the functional profile is used here). The first step is to associate each end-user function with a subinterval of the real interval $[0,1]$ whose size is equal to the input state's probability of selection $p(i)$.

Example:

Suppose that there are only three possible end-user functions, ADD, UPDATE, and DELETE. The functional profile indicates that the ADD function occurs 28% of the time, UPDATE occurs 11% of the time, and DELETE occurs 61% of the time. The ADD end-user function should be associated with the real interval $[0,0.28]$; the UPDATE function should be associated with the real interval $[0.28,0.39]$; and DELETE should be associated with the real interval $[0.39,1.0]$.

The next step is to generate a random number in the interval $[0,1]$ for each test case. Any number of short computer or programmable calculator programs are available that can generate random or pseudo-random numbers in that range.

Assume three test cases are to be performed. Three random numbers in the interval [0,1] are generated. The numbers are 0.7621, 0.5713, and 0.1499. Since the first random number, 0.7621, lies in the subinterval [0.39,1], the first test case is a DELETE. Since the second random number, 0.5713, also lies in the subinterval [0.39,1], the second test case is also a DELETE. Since the third random number, 0.1499, lies in the subinterval [0,0.28], the third test case is an ADD.

Testing efficiency can be increased by recognizing *equivalence classes*. An equivalence class is a set of input states such that if a run with one input state results in a failure, then, in theory, a run with any of the other input states in the class would also result in a failure. Conversely, if the program would succeed on a run with one input state in the class, then it would also succeed on any other input state in the class. Once an equivalence class is identified, only one representative input state from the class needs to be tested; if a run starting from the representative input state results in success, then it can be concluded that runs starting from all members of the class would result in success.

The input states that are members of an equivalence class are removed from the operational profile and replaced by their one representative input state. The probability associated with the representative input state is assigned the sum of the probabilities of the members of the equivalence class.

Since the probability of selection of the representative of an equivalence class is a sum, it can be relatively large compared to individual input states. The equivalence class representative input will likely be selected more than once during testing. After the first selection, the test case does not have to be re-run, only the results from the original run recounted.

The use of equivalence classes requires that the class developer(s) do a perfect job of creating the classes. In practice, this is not likely to ever be the case. This method, however, still provides an approximation to the operational profile that will reduce testing time significantly if the analyst does a reasonably good job of partitioning into equivalence classes.

8.3 Multiple Copies.

The time on test during growth or demonstration testing can be accumulated on more than one copy of the software. The copies can run simultaneously to accelerate testing. This procedure can be especially helpful in testing when the reliability requirement is very high. Because the total amount of calendar time on test is reduced, the use of multiple copies can provide economic and scheduling advantages. To retain the statistical integrity of the test, certain precautions must be taken.

Each copy must have its own separate data areas, both in main memory and secondary storage, to prevent cross-contamination. Each copy must use independently selected test inputs. The test inputs are selected randomly from the same operational profile. The time on test at any point in calendar time is the execution time accumulated on all versions. When one copy fails, it alone is recovered and restarted. If the processors on which the copies are running are of differing speed, the contributions to total time on test might need to be adjusted. For example, if the target processor in the operational environment has a speed of three million instructions per second (MIPS), and the three test processors

run at 4 MIPS, 2 MIPS, and 3 MIPS, respectively, then the first test processor's cumulative execution time must be multiplied by 4/3, the second processor's time must be multiplied by 2/3, and the third test processor's time requires no adjustment. This adjustment assumes that processor speed is the constraining factor on the system. That is, data is always ready to be processed.

Each tester should execute a set of test cases selected independently from the same operational profile. When a failure occurs on one copy, the execution time accumulated on all copies is recorded. When the program is repaired, all copies must be changed so as to remain identical.

8.4 Software Reliability Growth Modeling/Testing.

Reliability growth for software is the positive improvement of software reliability over time, accomplished through the systematic removal of software faults. The rate at which the reliability grows depends on how fast faults can be uncovered and removed. A software reliability growth model allows project management to track the progress of the software's reliability through statistical inference and to make projections of future milestones.

If the assessed growth falls short of the planned growth, management will have sufficient notice to develop new strategies, such as the re-assignment of resources to attack identified problem areas, adjustment of the project time frame, and re-examination of the feasibility or validity of requirements.

Measuring and projecting software reliability growth requires the use of an appropriate software reliability model that describes the variation of software reliability with time. The parameters of the model can be obtained either from prediction performed during the period preceding system test, or from estimation performed during system test. Parameter estimation is based on the times at which failures occur.

The use of a software reliability growth testing procedure to improve the reliability of a software system to a defined reliability goal implies that a systematic methodology will be followed for a significant duration. In order to perform software reliability estimation, a large sample of data must be generated to determine statistically, with a reasonable degree of confidence, that a trend has been established and is meaningful.

8.4.1 A Checklist of Software Reliability Growth Models.

There are several software reliability growth models available. Table 8-1 summarizes some of the software reliability models used in industry.

TABLE 8-1. Software Reliability Models¹

Model name	Formula for hazard function	Data and/or estimation required	Limitations and constraints
General Exponential (General form of the Shooman, Jelinski-Moranda, and Keene-Cole exponential models)	$K(E_0 - E_c(x))$	<ul style="list-style-type: none"> • Number of corrected faults at some time x. • Estimate of E_0 	<ul style="list-style-type: none"> • Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases linearly over time
Musa Basic	$\lambda_0[1 - \mu/v_0]$	<ul style="list-style-type: none"> • Number of detected faults at some time x (μ). • Estimate of λ_0 	<ul style="list-style-type: none"> • Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases linearly over time
Musa Logarithmic	$\lambda_0 \exp(-\phi\mu)$	<ul style="list-style-type: none"> • Number of detected faults at some time x (μ). • Estimate of λ_0 • Relative change of failure rate over time (ϕ) 	<ul style="list-style-type: none"> • Software must be operational. • Assumes no new faults are introduced in correction. • Assumes number of residual faults decreases exponentially over time
Littlewood/ Verrall	$\frac{\alpha}{(t + \Psi(i))}$	<ul style="list-style-type: none"> • Estimate of α (Number of failures) • Estimate of Ψ (Reliability growth) • Time between failures detected or 	<ul style="list-style-type: none"> • Software must be operational • Assumes uncertainty in correction process

¹For more information on these models see American Institute of Aeronautics and Astronautics, Recommended Practice for Software Reliability ANSI/AIAA R-013-1992, February 23, 1993; Farr, Dr. William, A Survey of Software Reliability Modeling and Estimation, NSWC TR 82-171, Naval Surface Weapons Center, Dahlgren, VA, Sept. 1983; Dr. Samuel Keene, G.F. Cole, Reliability Growth of Fielded Software, Reliability Review, Vol 14, March 1994; Musa, J.D., Iannino, A. and Okumoto, K., Software Reliability: Measurement, Prediction, Application. McGraw Hill Book Company, New York, NY. 1987.

		the time of the failure occurrence.	
--	--	--	--

TABLE 8-1. Software Reliability Models (Continued)

Model name	Formula for hazard function	Data and/or estimation required	Limitations and constraints
		•	•
Schneidewind model	$\alpha \exp(-\beta i)$	<ul style="list-style-type: none"> • faults detected in equal interval i • Estimation of α (failure rate at start of first interval) • Estimation of β (proportionality constant of failure rate over time) 	<ul style="list-style-type: none"> • Software must be operational. • Assumes no new faults are introduced in correction. • Rate of fault detection decreases exponentially over time
Duane's model	$\frac{\lambda t^b}{t}$	<ul style="list-style-type: none"> • Time of each failure occurrence • b estimated by $n/\sum \ln(t_n + t_i)$ from $i = 1$ to number of detected failures n. 	<ul style="list-style-type: none"> • Software must be operational
Brook's and Motley's IBM model	<p>Binomial Model</p> <p>Expected number of failures =</p> $\binom{R_i}{n_i} q_i^{n_i} (1 - q_i)^{R_i - n_i}$ <p>Poisson Model</p> <p>Expected number failures =</p> $\frac{(R_i \phi_i)^{n_i} \exp^{-R_i \phi_i}}{n_i !}$	<ul style="list-style-type: none"> • Number faults remaining at start of ith test (R_i) • Test effort of each test (K_i) • Total number of faults found in each test (n_i) • Probability of fault detection in ith test • Probability of correcting faults without introducing new ones 	<ul style="list-style-type: none"> • Software developed incrementally • Rate of fault detection assumed constant over time • Some software modules may have different test effort than others
Yamada, Ohba, and Osaki's S-Shaped model	$ab_2 t \exp^{-bt}$	<ul style="list-style-type: none"> • Time of each failure detection • Simultaneous solving of a and b 	<ul style="list-style-type: none"> • Software is operational • Fault detection rate is S shaped over time

TABLE 8-1. Software Reliability Models (Continued)

Model name	Formula for hazard function	Data and/or estimation required	Limitations and constraints
Weibull model	$MTTF = \frac{b}{a} \Gamma\left(\frac{1}{a}\right)$	<ul style="list-style-type: none"> • Total number faults found during each testing interval • The length of each testing interval • Parameter estimation of a and b 	<ul style="list-style-type: none"> • Failure rate can be increasing, decreasing or constant
Geometric model	$D\phi^{i-1}$	<ul style="list-style-type: none"> • Either time between failure occurrences X_i or the time of the failure occurrence • Estimation of constant D which decreases in geometric progression ($0 < \phi < 1$) as failures are detected. 	<ul style="list-style-type: none"> • Software is operational • Inherent number of faults assumed to be infinite • Faults are independent and unequal in probability of occurrence and severity
Thompson and Chelson's Bayesian Model	$\frac{(f_i + f_0 + 1)}{(T_i + T_0)}$	<ul style="list-style-type: none"> • Number of failures detected in each interval (f_i) • Length of testing time for each interval i (T_i) 	<ul style="list-style-type: none"> • Software is corrected at end of testing interval • Software is operational • Software is relatively fault free

The following checklist determines which model or models to choose from given the following constraints. This checklist is summarized in Figure 8-1.

- Failure profiles
- Maturity of software product
- Characteristics of software development
- Characteristics of software test
- Existing metrics and data

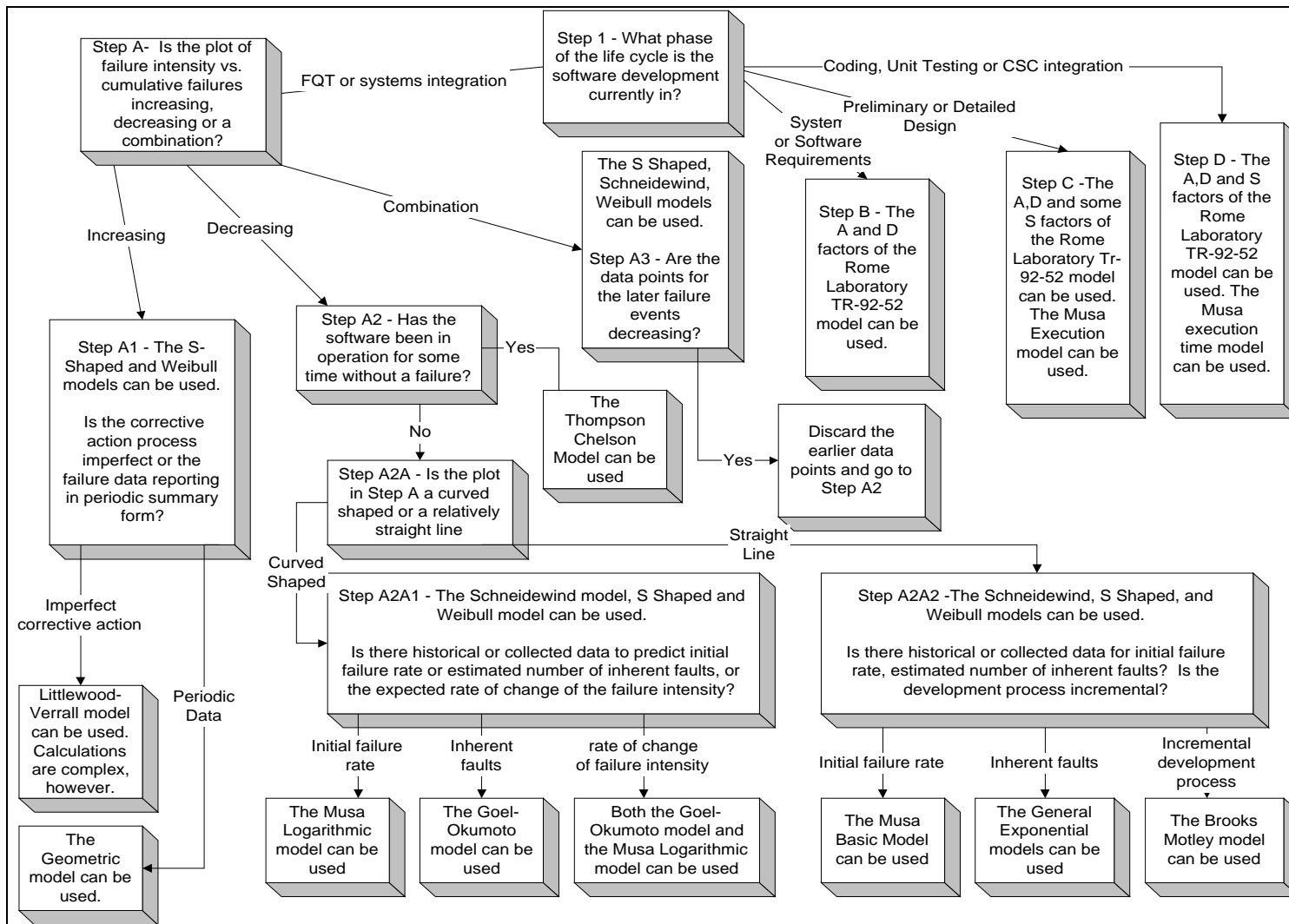


FIGURE 8-1. Software Reliability Growth Models

Step 1 - What is the current software life cycle phase?

If the software is currently in Formal Qualification Test or System Integration -

Step A - Is the plot of cumulative unique failures per cumulative test time (failure intensity) versus cumulative failures detected or versus cumulative time, going from the top left hand corner to bottom right hand corner of graph? Is there a positive y intercept for either one of these graphs? In other words, is the failure intensity increasing or decreasing or a combination of increasing and decreasing with respect to either time or cumulative failures detected?

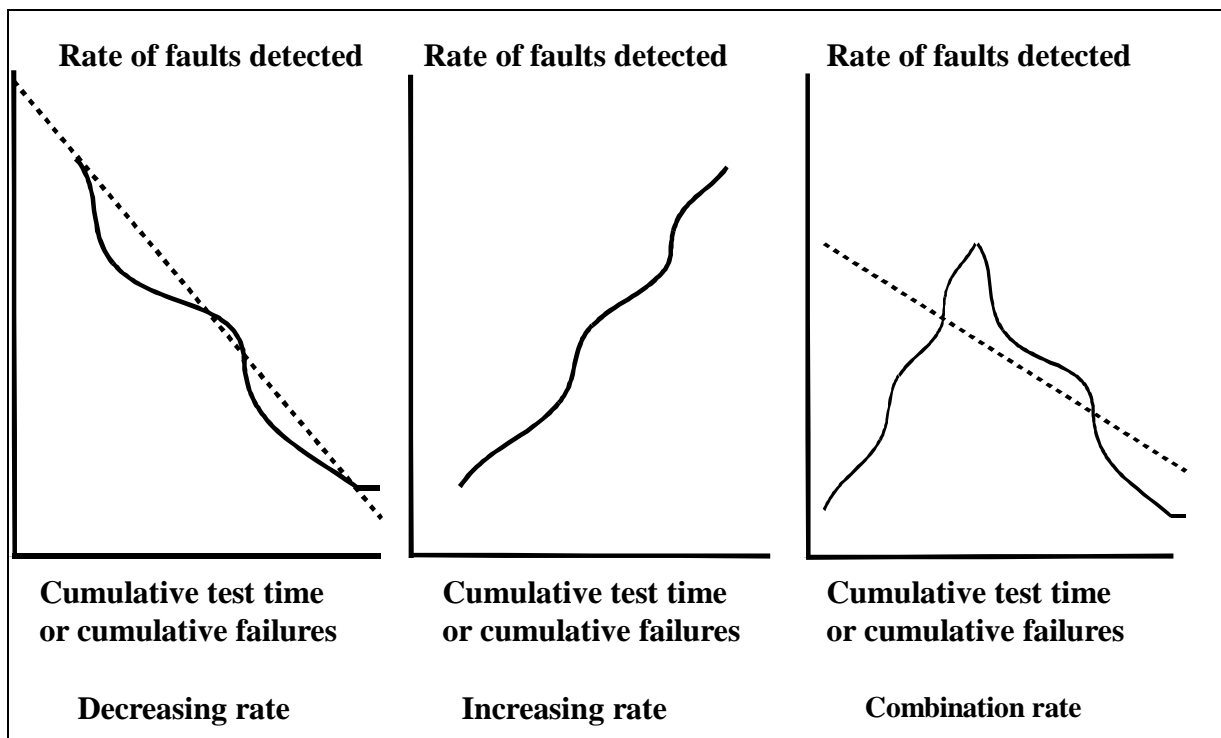


FIGURE 8-2 Failure Rate Profiles

Step A1 - If increasing,

1. Make sure that the software is in an operational state.
2. Make sure that only unique software failures are being counted.
3. Make sure that estimate of time is accurate.

4. If all of the above check out, then the software is likely to be in an early stage of system testing and/or development.

It is possible that the S-Shaped and Weibull models can be used. If it appears that there is an imperfect corrective action process then use the Littlewood-Verrall model. If the test results are in periodic summary form then use the Geometric model.

Step A2 - If the failure intensity is decreasing, has the software been tested or used in an operational environment representative of its end usage, with no failures for a significant period of time?

If yes, the Thompson Chelson model can be used.

Otherwise go to Step A2A.

Step A2A - Does the plot discussed in step A better represent the curved shape failure intensity in the first picture or any of the curves in the second picture?

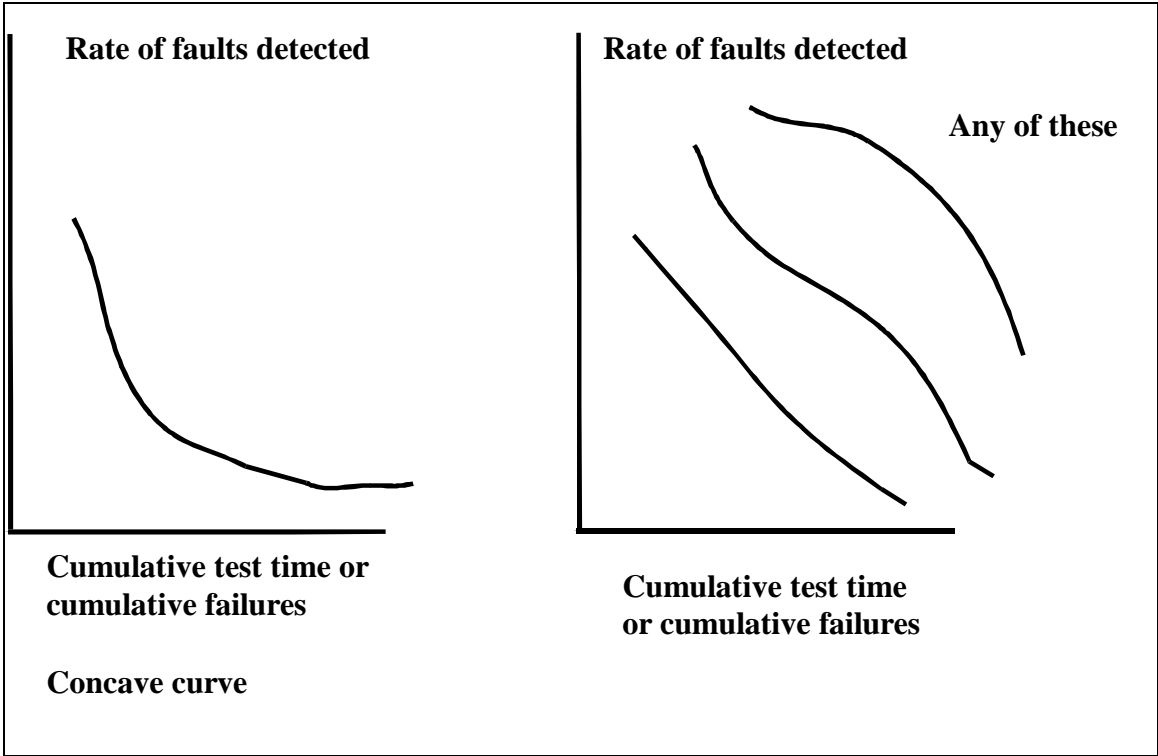


Figure 8-3 Failure Rate Curves

Step A2A1 - If there is a curved failure rate profile, the Schneidewind, Weibull, Musa logarithmic model or the Goel-Okumoto models can be used.

If historical data on the failure rate at the start of system testing is available, then use the Musa logarithmic model. If historical data on the estimated number of inherent faults or the expected rate of faults detected per time is available, then use the Goel-Okumoto model or the Musa logarithmic model.

The Schneidewind and Weibull models can be used, but require estimation of two unknowns using simultaneous equations.

Step A2A2- If any of the straight line profiles more closely represent the plot, then the Musa Basic, general exponential, Schneidewind, Weibull, and Brooks-Motley models can be used.

If there is historical data on the failure rate at the start of system testing then the Musa Basic model can be used.

If there is historical data on the estimated number of inherent faults or the expected rate of faults detected per time, then the general exponential model can be used.

If the software is being tested and/or developed incrementally the Brooks Motley model can be used.

The Weibull, S-Shaped and Schneidewind models can be used, but require estimation of two unknowns using simultaneous equations.

Step A3 - If the plot has a combination of increasing and decreasing failure intensities then the S-Shaped, Schneidewind, and Weibull models can be used. Also, if the data points which are increasing are earlier data and the later data points are decreasing over the x axis, then the earlier data points can be discarded and a model selected by going to step A2. Note that in order to discard the earlier data points, there should be a relatively significant number of data points that are decreasing.

Step B - Is software development effort currently in the requirements phase?

It is too early to perform reliability estimations or growth assessments. The predictive Rome Laboratory RL-TR-92-52 Software Reliability Measurement and Test Integration Techniques model can be used by solving for just the A and D factors. See Section 7 for a description of predictive models.

Step C - Is the software in the design phase?

It is still too early to perform reliability estimations or growth assessments. The predictive Rome Laboratory RL-TR-92-52 Software Reliability Measurement and Test Integration Techniques model can be used by solving for the A, D and any of the S factors that are known. The Musa execution time model can be used to predict initial failure rate at start of testing. See Section 7 for other predictive techniques that can be used in this phase.

Step D - Is the software in the coding phase, unit testing phase or CSC integration phase?

It is too early to perform system reliability estimations or growth assessments. However, results from unit testing can be tracked to derive an estimate of the number of *functional* faults that the software contains. This can be combined with the predictive Rome Laboratory RL-TR-92-52 Software Reliability Measurement and Test Integration Techniques model. The Musa execution time model can also be used to predict initial failure rate at start of testing. See Section 7 for other predictive techniques that can be used in this phase.

8.4.2 Goodness-of-Fit/Recalibration.

When the time-domain software reliability models are employed, it is not sufficient to blindly apply the model. The tester should monitor how well the model is fitting the failure data. Figures 8-2 and 8-3 in Section 8.4-1 are intended to be a guide for making sure the model fits the data.

If the model is not fitting well, then the user should switch to an alternative model and/or parameter estimation technique. Some software reliability modeling tools allow models to be combined, or to develop your own model. Another option is to employ a technique known as adaptive reliability modeling or recalibration. This technique uses the data on the historical performance of the software reliability model on the program in question to modify the model itself. The accuracy of recalibrated models has been shown generally to be better than that of the original model. See the appendix for more information on this technique.

8.4.3 Collecting the Data Required for the Models.

There are a variety of tools available for collecting the data necessary for using any or all of the models discussed in the previous section. The tool used should be capable of tracking fault events as well as plotting and projecting reliability growth. Generally, most of the effort required is to implement and/or automate one model. Automating additional models typically requires less relative effort. It is a good idea to be prepared to automate/implement at least one model from each category of models:

- Models that assume fault detections are uniform over time
- Models that assume some faults are more likely to occur than others
- Models that take into account mixed fault profiles

8.5 Software Reliability Demonstration.

The purpose of reliability demonstration testing is to verify, with a stated degree of statistical confidence, that the system or the software product meets the specified reliability requirement.

An executing program can be modeled as having a constant failure rate when its code is frozen and it is being subjected to inputs randomly selected from a stationary operational profile. An operational profile associates each possible input state with a probability of selection. A *stationary* operational profile means that those probabilities stay the same throughout the test period.

The ideal operational profile is the one the software system will experience in its intended operating environment, because then the system's failure behavior will be representative of what the end-user would experience if the program were released.

The best source of data to determine occurrence probabilities is historical data consisting of usage measurements taken on a latest release, a similar system, or a manual function that is being automated.

There are two ways to determine occurrence probabilities for operations: (1) record the input states in the field, group them into operations, and count them; or (2) rely on estimates derived by refining the functional profile. The first is more accurate, but can be done only if a previous release exists.

Recording. It may take some effort to develop recording software, but the benefits for the application and, perhaps more important, for the company as a whole, can far outweigh the effort. It may be possible to develop a generic recording routine that requires only an interface to each application. The recording software must instrument the system so that it extracts sufficient data about input variables to identify the operations being executed. Then the task reduces to simply counting the execution of each operation. An operational profile may then be recorded explicitly.

Because the operational profile used in test may not match the true operational profile in the field, it is called the *test operational profile*. In this case, an appropriate transformation will be required to convert failure intensity experienced under this profile to what would occur in the field. This will not be feasible until after the software is fielded.

In general, the following types of hardware are modeled by a constant failure rate: (1) parts that are in their "useful life" period, which is after burn-in but before wearout; (2) assemblages of those parts, when in a series reliability configuration; and (3) complex, maintained equipment that does not have redundancy. When software runs concurrently and in series with such hardware, the overall failure rate will be a constant that is the sum of the constant hardware failure rate and the constant software failure rate. In the remainder of this section, the item under test will be referred to as "software," but the term should be understood as applying to both software products and to combined hardware/software systems in which the hardware can be modeled by a constant failure rate.

Let λ be the true failure rate of the software. In designing a demonstration test, two failure rates, λ_0 and λ_1 , must be specified ($\lambda_0 < \lambda_1$). A good test plan will reject, with high probability, software with a

true failure rate that approaches λ_1 . A good test plan will accept, with high probability, software with a true failure rate that approaches λ_0 .

Relying on the results of the demonstration test for making an accept/reject decision entails two basic risks. First, if good software happens to perform poorly (fails too many times during the test), then it could be rejected. Conversely, if a low quality software configuration item performs well during the test, fault-laden software could be accepted. These two risks must be specified in advance as parameters to the test. The *producer's risk* is the probability of rejecting software with a true failure rate equal to λ_0 . The *consumer's risk* is the probability of accepting software with a true failure rate greater than or equal to λ_1 .

Three types of demonstration tests are recommended for software: fixed duration test, failure-free execution interval test, and sequential test.

A fixed duration test is used when the amount of test time and cost must be known in advance. A fixed duration test provides demonstrated failure rate to a desired confidence level.

A sequential test will accept software that has a failure rate much lower than λ_0 and reject software that has a failure rate much higher than λ_1 , more quickly than a fixed duration test having similar parameters. However, the total test time may vary significantly according to the true failure rate.

A failure-free execution interval test will accept software that has a failure rate lower than λ_0 more quickly than a fixed duration test.

Producer's and consumer's risks usually range from 10% (low risk) to 30% (high risk). The lower the risks, the longer the test. The ratio

$$\delta = \frac{\lambda_1}{\lambda_0} \quad (8.2)$$

is called the *discrimination ratio*. The discrimination ratio establishes the *power* of the demonstration test in distinguishing between reliable and unreliable items. The lower the discrimination ratio, the more test time required. The simultaneous execution of multiple copies will allow a lower discrimination ratio or save test time. Some low failure rates may be impossible to demonstrate without using multiple copies.

The standard fixed duration and sequential test can be used for software because these plans assume a constant failure rate (exponential time-to-failure distribution). These test plans are parameterized in terms of Mean Time to Failure (MTTF).

If it is assumed that the program cannot be perfect fault-free, then the MTTF θ will always exist. The MTTF is the reciprocal of the failure rate:

$$\theta = \frac{1}{\lambda} \quad (8.3)$$

The higher the MTTF the greater the software reliability. The *lower test MTTF* is $\theta_1 = 1 / \lambda_1$. The *upper test MTTF* is $\theta_0 = 1 / \lambda_0$. To use the tests, one must specify α , β , and δ . The tabulated values of the decision risks are 10%, 20%, and 30%. The available values for δ are 1.5, 2.0, and 3.0. Not all combinations appear.

In a failure-free execution period test, the software is given T time units to achieve a failure-free interval of t time units. In the most stringent version of this test, $t = T$; the software must get through the test with zero failures. In the zero-failure test, the discrimination ratio will be $\ln \beta / \ln (1-\alpha)$ and the test time will be $t = T = - \ln \beta / \lambda_1 = - \ln (1 - \alpha) / \lambda_0$. For lower discrimination ratios, tests can be designed in which $t < T$. Since β is small, the test time will tend to be just a little longer than θ_1 .

The parameters of the test are the α and β risks, λ_0 , and λ_1 . Table 8-2 provides test plans for various combinations of α , β , and δ . The test time T is obtained by dividing either column 4 by λ_1 or dividing column 5 by λ_0 . Once T is obtained, the duration t of the failure-free interval is obtained by multiplying column 6 by T. The Expected Test Time (ETT) depends on what the true failure rate is. The true failure rate is not known. The ETT when the true failure rate is λ_1 is obtained by multiplying column 7 by T. The ETT when the true failure rate is λ_0 is obtained by multiplying column 8 by T.

Before the test, clear definitions must be established as to what constitutes satisfactory operation and what constitutes failure. The definitions need to be agreed on by both developer and customer.

The software environment must emulate the field operating conditions. These include the operating system version and the versions of other system software with which the application will interact, system parameter settings, workload (transactions per second, for example), as well as the operational profile. The hardware environment should be the actual hardware used in the field. To maintain the statistical integrity of the test, each failure must be statistically independent of the others. Thus, when a failure occurs, any corrupted files or databases must be restored and the software restarted.

TABLE 8-2. Failure-Free Execution Interval Test Plans

1	2	3	4	5	6	7	8
α	β	δ	$\lambda_1 T$	$\lambda_0 T$	t/T	ETT/T λ_1	ETT/T λ_0
.10	.10	2.442	63.308	25.925	.10	.88	.43
.10	.10	2.814	38.581	13.710	.15	.84	.45
.20	.20	1.793	54.330	30.301	.10	.84	.52
.20	.20	1.968	32.618	16.574	.15	.81	.53
.20	.20	2.147	22.445	10.454	.20	.78	.54
.20	.20	2.338	16.640	7.117	.25	.76	.55
.20	.20	2.547	12.927	5.075	.30	.73	.56
.20	.20	2.779	10.365	3.730	.35	.71	.58
.20	.20	3.052	8.501	2.785	.40	.68	.59
.30	.30	1.438	48.707	33.871	.10	.80	.59
.30	.30	1.695	14.361	8.473	.25	.74	.61
.30	.30	1.995	7.088	3.553	.40	.68	.62
.30	.30	2.454	4.086	1.665	.55	.62	.63
.30	.30	3.059	2.526	.826	.70	.58	.66

For the purposes of demonstration testing, only one software failure can occur per run. This is different from growth testing where every discrepancy between the actual values of output variables and their required values is counted as a separate failure, if caused by a different fault.

Procedure 8.5-1 - Demonstration test.

Steps.

- A. Obtain the specification of the lower test MTTF θ_1 , an unacceptable value.
- B. Obtain the consumer's risk, the probability of accepting software whose true MTTF is θ_1 .
- C. Obtain the value of the upper test MTTF θ . This is the goal the producer attempted to achieve.

- D. Obtain the producer's risk, the probability of rejecting software whose true MTTF is θ_0 .
- E. Calculate the discrimination ratio as θ_0/θ_1 .
- F. Choose a fixed length or sequential test plan, or choose a failure-free execution interval test plan from Table 8-2.

Example 1 - Fixed-length test plan:

The customer specifies the lower test MTTF θ_1 as 500 hours. The producer's and consumer's risks are set at 20%. The reliability goal for the software was specified as 750 hours to failure. Design a fixed duration test.

- A. The discrimination ratio is calculated at $750 / 500 = 1.5$.
- B. The duration of the test is provided as 215×500 hours = 10,750 hours. The acceptable number of failures, therefore, is 17 or fewer.

Example 2 - PRST test plan:

The customer specifies the lower test MTTF θ_1 as 600 hours. The producer's and consumer's risks are set at 10%. The reliability goal for the software was specified as 1200 hours. Design a PRST test plan.

The discrimination ratio is calculated as $1200 / 600 = 2.0$. The minimum time to accept decision is 4.40×600 hours = 2640 hours. The expected time to an accept decision (assuming a true MTTF equal to θ_0) is 10.2×600 hours = 6120 hours. The maximum time to reach an accept decision (assuming a true MTTF equal to λ_1) is 20.6×600 hours = 12360 hours.

Example 3 - Failure-free execution interval test plan:

The customer specifies λ_1 as 0.0001 failures/hour. The producer's and consumer's risks are set at 30%. The reliability goal for the software was specified as $\lambda_0 = 0.00005$ failures/hour.

The discrimination ratio is calculated as $0.0001/0.00005 = 2.0$. Entering Table 8-2 at $\alpha=.30$ (column 1), $\beta=.30$ (column 2), and $\delta=1.995$ (column 3) provides $\lambda_1 T = 7.088$ (column 4), or $T = 70880$ hours. Since $t/T = .40$ (column 6), $t = 28352$.