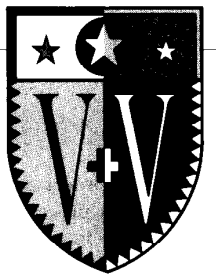

Quantifying Software Validation: When to Stop Testing?

John D. Musa, *AT&T Bell Laboratories*
A. Frank Ackerman, *Institute for Zero-Defect Software*

System testing becomes guesswork — unless you set it up to apply statistical analysis. Then you can focus attention on the software's use instead of its structure.

How do you validate that a piece of software loaded into a processor functions correctly? One traditional answer is that you subject it to a rigorous system test. But there is a fundamental problem: For any but the most trivial application, the number of distinct input combinations you would need to verify is enormous — orders and orders of magnitude larger than any number that can be tested exhaustively.

Furthermore, because of the discrete nature of computer memory and processing, the difference of a single input bit out of thousands may be all that separates an input combination that runs successfully from one that doesn't.

How then do you validate software? In hard engineering terms, the answer is that up to now you really haven't. There is a lot of lore about system testing, but it all boils down to guesswork. That is, it is guesswork unless you can structure the problem and perform the testing so that you can apply mathematical statistics.

If you can do this, you can say something like "No, we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95-percent confidence that the probability of 1,000 CPU hours of failure-free operation in a probabilistically defined environment is at least 0.995."

When you do this, you are applying *software-reliability measurement.* In this situation, this is the *best* you can do. For purists, this may not be a satisfactory answer to our initial question. But with software-reliability measurement, you do not deal explicitly with the vastness, discreteness, and discontinuity of a program input space — you sidestep these imponderables by using statistics to provide concrete, quantitative guidance.

In this article, we define the basic concepts of software-reliability measurement

0740-7459/89/0500/0019/$01.00 © 1989 IEEE

and show you how to use them in software validation.

Basic concepts

Consider some basic concepts of software reliability:

• Run. Many kinds of software run continuously. Two examples are process-control systems and telephone-switching systems. Others, like automatic teller machines and electronic mail, run on demand. In either case, you can break down the ongoing operation of the software into a series of discrete runs. Each run performs a mapping between a set of input variables and a set of output variables and consumes a certain amount of processor execution time. The specifications of this mapping are the requirements for the run.

• Software failures and faults. When a run is made where the outputs do not conform to the requirements, one or more *failures* have occurred. To prevent a particular failure from occurring on a subsequent run, you must modify the program by changing the set of instructions that it executes — you must correct the *fault* that underlies the failure.

This is a very general definition of failure. It covers not only departures from desired functionality but also performance issues. For example, if a response to a given input is required for a specified hardware configuration and load within a specified time and if this response is not made within that time, a failure has occurred.

• Operational profile. The exact inputs that are presented moment by moment to an operational piece of software must be considered to be selected at random. But some inputs will occur on the average more frequently than others. The specification of these inputs and their relative frequencies is called the program's operational profile.

• Software failure occurrence is a Poisson process. Given a very general set of assumptions, you can show that software failure occurrence obeys a Poisson process.[1] This is expected, since other similar phenomena — like the number of phone calls received at a switchboard in a given period or the number of taxis that break down each day in a given city — are also modeled by Poisson processes. You can characterize a Poisson process by its expected value function. In reliability measurement, this is the cumulative number of failures, $\mu(\tau)$, expected to occur by the time the software has experienced a given amount of execution time, τ.

Execution time is the basic dimension of reliability measurement because, given a random selection of inputs based on an operational profile, execution time accurately reflects software stress: A piece of software that is never executed never fails. The use of execution time yields reliability measures that are relative to the speed of the processor executing the software.

Given that a Poisson process models failure occurrences in general, you provide models for specific situations by specifying the function $\mu(\tau)$. Several functions have been considered in the literature, but we will describe only three because they model most situations encountered in practice. They are simple to use and provide descriptive and predictive accuracy commensurate with the other elements of current software-engineering practice. Figure 1 illustrates these functions.

Execution time is the basic dimension of reliability measurement because it accurately reflects software stress: A piece of software that is never executed never fails.

• Modeling failure occurrence for unchanging software (Figure 1a). The first function is $\mu(\tau) = \lambda\tau$, where λ is a constant. This function models the case where both the software and its operational profile remain unchanged. A good example is computer-terminal software, which is usually permanently installed in a terminal as firmware.

This function simply says that the expected number of failures that will occur after τ units of execution time (measured over all processors running this software) is directly proportional to τ. The instantaneous rate of failure (that is, the derivative of μ), which is denoted by $\lambda(\tau)$, is an important function in reliability measurement. It is called the failure-intensity function. In this case, $\lambda(\tau)$ equals λ. We call this the *static* execution-time model.

• Modeling failure occurrence for software being debugged (Figure 1b). During system test, the faults revealed by test failures are constantly being corrected, so failure intensity should decrease as testing proceeds. If the faults in the software are equally likely to cause failures so that the average failure intensity improves by the same amount whenever a correction is made, you can show that

$$\mu(\tau) = \nu_0\left[1-\exp\left(-\frac{\lambda_0}{\nu_0}\tau\right)\right]$$

This function models the case where the operational profile remains unchanged but where an action (possibly imperfect) is initiated to correct the responsible fault whenever a software failure occurs. (Correction need not be immediate. If it isn't, we simply do not count repeated occurrences of the same failure.)

For this model,

$$\lambda(\tau) = \lambda_0 \exp\left[-\frac{\lambda_0}{\nu_0}\tau\right]$$

We call this the *basic* execution-time model.

Although the assumptions on which the basic execution-time model are based may not always be satisfied in practice, the model still provides a reasonable representation and usually provides useful results. The major advantage of this model is that its parameters — λ_0 (the initial software failure intensity at the beginning of the observation period) and ν_0 (the total number of failures that will be experienced in an infinite amount of execution time) — can both be easily related to other attributes of the software, to its execution environment, and to the development process.

• If you take the converse of one of the assumptions that led to the previous model — if you assume that some faults are more likely to cause failures and that on the average the improvement in fail-

ure intensity with each correction declines exponentially as corrections are made — you can show that

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1)$$

Like the previous function, this function also models the situation where the operational profile remains unchanged and where corrections (possibly imperfect) are made whenever a failure occurs. For this model,

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1}$$

We call this the *logarithmic Poisson* execution-time model (Figure 1c).

The parameter λ_0 of the logarithmic Poisson model has the same meaning as it did in the basic model (but it will usually have a different value). The parameter θ characterizes the exponential drop-off in failure intensity as repairs are made. The assumption about the relationship between faults and failures on which the logarithmic model is based is particularly suited to when the operational profile is highly nonuniform.

System testing

Applying reliability measurement during system test provides quantitative information on the validation process. System testing has two distinct functions:

- to validate that the completed software functions close enough to its requirements to warrant passing it on to the next stage in the life cycle and
- to improve the quality of the completed software by first discovering the input states on which it fails to meet requirements and then eliminating these failures by correcting the underlying faults.

Reliability measurement is concerned mostly with the first function. The discovery of failure input states is outside reliability measurement's domain, but measurement can help you concentrate testing attention through the concept of the operational profile.

Presuppositions. The use of reliability measurement during system test presupposes that

- the definition of significant failures has been specified,
- an operational profile has been specified, and
- for the defined failures and operational profile, a failure-intensity objective has been set to some desired level of confidence.

The first presupposition can be met simply by defining a departure from any of the functional requirements as a failure. However, some types of failures are usually more significant than others; you handle this reality by defining failure-severity classes. Ideally, all these presuppositions would be clearly defined in the quality-

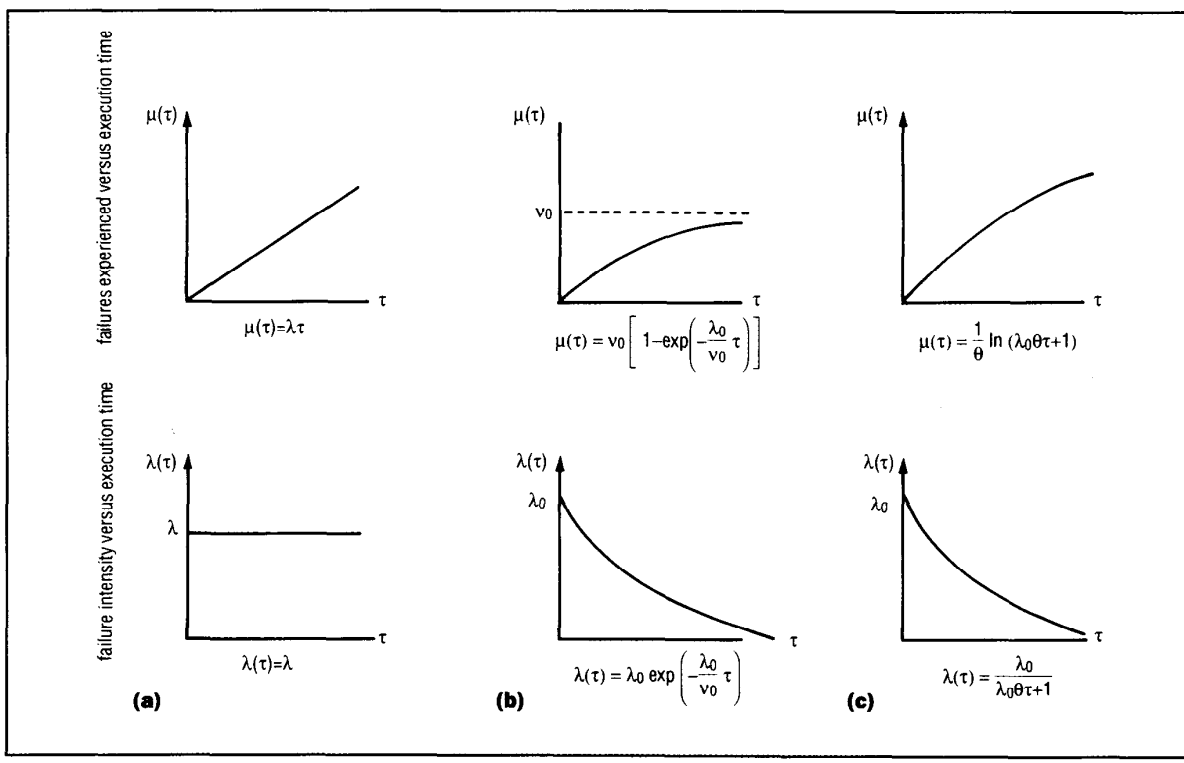

Figure 1. Three useful software-reliability models: **(a)** static, **(b)** basic, and **(c)** logarithmic Poisson. These are shown comparing both failures experienced versus execution time and failure intensity versus execution time.

Table 1.
Inputs to a reliability-measurement program

Parameter data		Failure data		
Parameter	Value	Failure number	CPU seconds from last failure	Day of system test
Model type	Basic execution time	1	3	1
		2	30	2
Failure-intensity objective	0.01 failures per CPU hour	3	113	9
		4	81	10
Test-compression factor	2.0	...	...	...
		97	1,261	73
System-test resource parameters	various values	98	1,800	73
Current date	05/03/89			

attributes section of the requirements specification you use.[2]

The use of reliability measurement during system test also presupposes that you can directly relate the occurrence of failures to the amount of execution time the software has experienced when failures occur — you need to be able to plot μ against τ. Often, you cannot measure the execution time directly. While you can use a hardware analyzer to measure execution time directly if you are validating software for a small system, you tend to measure execution time indirectly for larger, more complex systems. One such indirect method is to determine the average amount of execution time used per command or transaction and then count the commands executed to determine execution time.

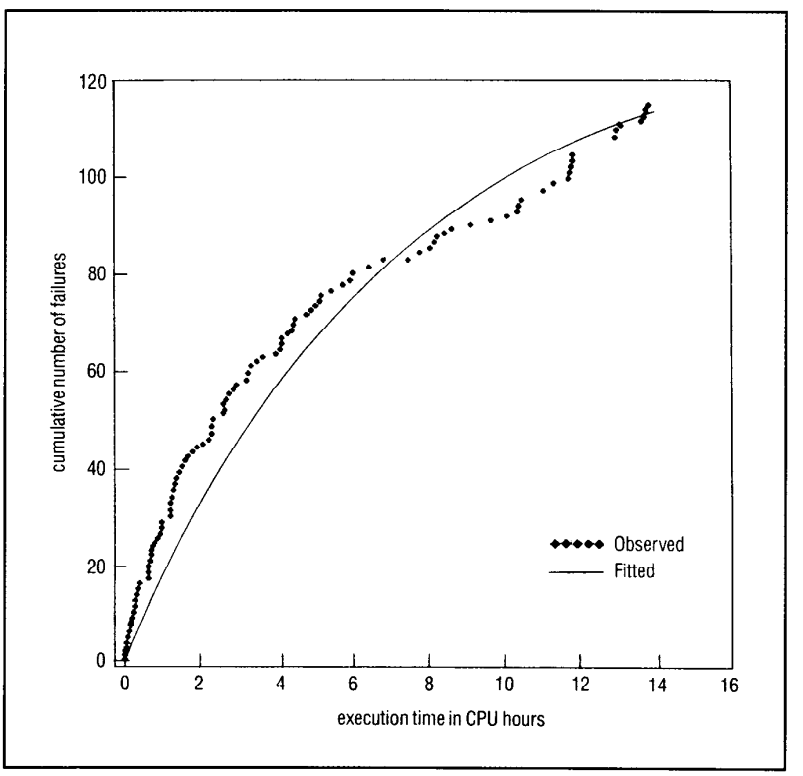

Figure 2. Observed failures and the basic model.

Basic procedure. With these presuppositions in place, all that is necessary to apply reliability measurement during system test is to

- select test cases in proportion to the frequencies specified by the operational profile,
- record, at least approximately, the amount of execution time between failures, and
- continue testing until the selected model shows that the required failure-intensity level has been met to the desired level of confidence.

Table 1 and Figures 2-4 illustrate this procedure. Table 1 shows the input to a program that implements the basic and the logarithmic Poisson models described earlier and that performs the statistical confidence-interval calculations. Figure 2 shows a plot of failure data similar to that in Table 1 and a basic execution time function that fits this data (using maximum-likelihood estimation). This fit provides values for the model parameter v_0 and λ_0. You can then use these parameters to calculate the other values shown in Figure 3, which illustrates the use of reliability measurement as a verification and validation tool.

At this point in system testing, you have established with 95-percent certainty that the failure intensity of this system is not greater than 1.467 failures per CPU hour. Furthermore, if subsequent test results follow this pattern, you can be 95-percent certain that you will reach your release objective of 0.05 failures per CPU hour in less than 10.2 CPU hours of testing and that during this period you will need to address the faults responsible for no more than 14 new failures. Figure 4 shows how failure intensity changes with corrections for each failure.

The procedure illustrated in Table 1 and Figures 2-4 requires that enough failures are being observed to obtain statistically valid results. In general, this means that the individual system components to which reliability measurement is applied must be reasonably large, say 5,000 lines of code when the initial fault density is five per thousand lines. Given components of at least this size, a test manager can use reliability measurement to allocate test resources among these components. For ex-

ample, components with higher-than-average initial failure intensities will require more testing time to bring them to the same reliability levels than components with lower-than-average initial failure intensities.

Testing compression. Figures 2 and 3 also illustrate another concept: testing compression. If you select test cases strictly at random according to the operational profile, the failure-intensity level computed at the completion of system testing should match the static level observed later in the field. But — given the need to conserve system-testing resources and the need to accomplish the two objectives of validation and quality improvement in the least amount of time — software system testing, like its hardware counterpart, seeks to accelerate stress. In testing compression, you select test cases so that repetition is limited, thus eliminating redundancies that are unlikely to lead to new failures and thus speeding up reliability improvement.

The project illustrated in Table 1 has a testing-compression factor of two, so the failure-intensity values expected in operation are half those measured in test and the desired failure-intensity objective is reached sooner.

Calendar-time component. The last two lines of the output shown in Figure 3 are the result of translating CPU time into project time (given input about the project's system test resources) to predict the actual calendar date on which testing will be complete. This calculation requires a separate calendar-time component, which is fully described in the book by Musa, Iannino, and Okumoto.[1]

With this component, the system-test manager can investigate the effect of rearranging system-test resources on the completion date. For example, you might investigate using additional computer time for system testing or you might plan for more system testers. Conversely, if you are faced with a fixed release date and a fixed quantity of testing resources, you can use the program in Figure 3 to predict the failure intensity that can be expected at the end of the predefined test period. Project managers can then use this information to adjust schedules and resources or to renegotiate them with the customer.

```
                SOFTWARE RELIABILITY PREDICTION
                        BASIC MODEL
                        SYSTEM T98c2

BASED ON SAMPLE OF    98 TEST FAILURES
EXECUTION TIME IS     11.03 CPU HR
FAILURE-INTENSITY OBJECTIVE        50 E-01 FAILURE/CPU HR
CALENDAR TIME TO DATE IS 73 DAYS
CURRENT DATE:  05/03/89

    CONF. LIMITS              MOST              CONF. LIMITS
  95%   90%   75%   50%    LIKELY   50%   75%   90%   95%

TOTAL FAILURES
  101.  101.  102.  103.  ❶105.    107.  110.  113.  115.

FAILURE INTENSITIES (FAILURE/1000 CPU HR)

INITIAL F.I.
  9884 10443 11181 11901 ❷12967 14054 14837 15663 16194

CURRENT F.I.
 469.7 517.6 601.8 694.2  846.6  1027  1175  1347  1467

❸ *** ADDITIONAL REQUIREMENTS TO MEET F.I. OBJECTIVE ***

FAILURE
     1     2     2     3      5     7     9    12    14

EXECUTION TIME (CPU HOURS)
  1.97  2.35  3.02  3.75   4.93  6.37  7.58  9.07  10.2

CALENDAR TIME (DAYS)
  .932  1.18  1.65  2.22   3.28  4.76  6.16  8.04  9.53

COMPLETION DATE
 41489 41589 41589 41689  41789 42089 42289 42489 42789
```

Key
❶ Estimate for ν_0
❷ Estimate for λ_0 (12.97 failures per CPU hour)
❸ These quantities are calculated by using the parameters just estimated to calculate the points at which the failure-intensity objective will be reached.

Figure 3. Annotated tabular output of a reliability-measurement program.

Requirements definition

Although the application of reliability measurement to data from observed failures must wait until system-test or later phases, software-reliability engineering nonetheless has important contributions to make early in the project. For example, during requirements definition, you must determine the system-test failure-intensity objective and the amount of system-test resources required to reach this objective.

To define failures and the operational profile for the system, engineers must extend their dialogue with customers and users. Special studies of existing systems may be needed to determine operational profile frequencies, but the additional system-specification effort required to apply reliability measurement can have an immediate benefit.

First, customers and users will become aware that, *within* a given development technology, higher levels of certified quality require more testing, which will be reflected in higher costs at some point. With reliability measurement, you can speak quantitatively and concretely about quality levels and thus about costs. Given the cost constraints, customers and users will have to make up-front choices and communicate these choices to your system en-

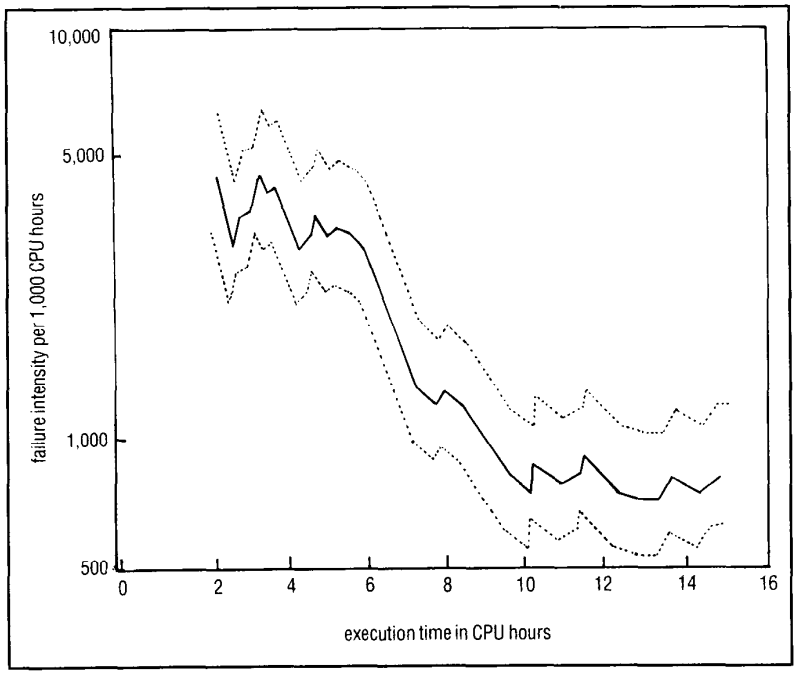

Figure 4. Graphical output of a reliability-measurement program.

gineers.

Second, the definition of the operational profile will provide valuable information on how the users expect to use the software. The definition of failures is useful in clarifying the requirements. The additional information required for reliability measurement will mean a better requirements specification and more complete information on which you can base design and implementation decisions.

Ultimately, you must take into account *all* of the costs associated with a chosen failure-intensity level. For the two reliability-growth models we have presented, system-test cost increases nonlinearly with increasing quality (that is, with decreasing failure intensity). On the other hand, for a given life (in total execution time across all installations), the cost of cleaning up after a failure or of making warranty payments for failure will generally increase with decreasing levels of quality — the higher the quality, the lower the postdelivery costs, as Figure 5 shows. The figure suggests that you should select an optimal failure-intensity objective that will minimize the total life-cycle cost of failure.

You can estimate the amount of system testing you will need to reach a specified failure-intensity objective if you can determine the parameters for the appropriate reliability-growth model. Such values must be based on data from experience. Ideally, such data would come from quality-assurance information about an earlier release of the software, from releases of similar software, or from the development organization's average experience. Lacking any of these more desirable sources, you can take parameter values from averages of published data. You can likewise get the equations needed to predict the parameters λ_0 and ν_0 for the basic model from published sources.

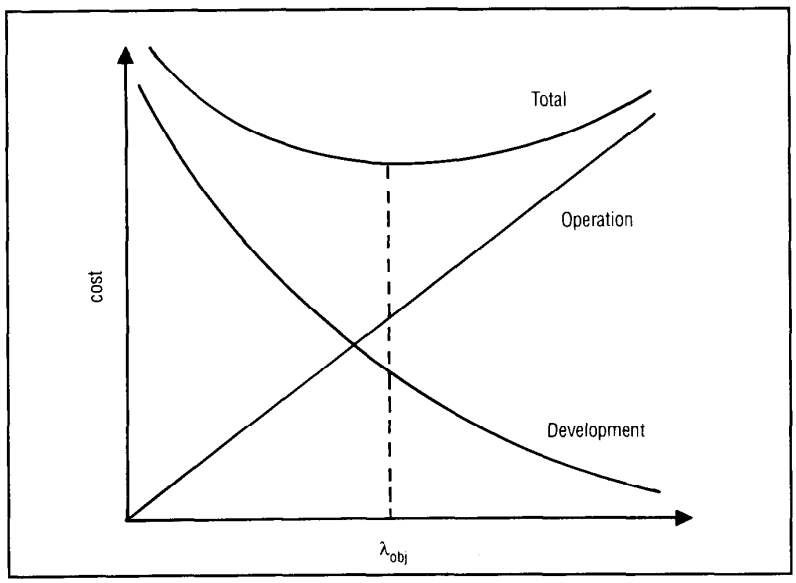

Figure 5. Selecting a failure-intensity objective that will minimize the cost of failure over the life cycle.

Acceptance testing

Suppose you receive a piece of software from a supplier who claims this software meets your reliability requirements. How can you validate this claim? The answer is to use an acceptance chart based on a sequential sampling technique.

Because failure occurrence is a Poisson process, you can convert from reliability to failure intensity by using $R=e^{-\lambda\tau}$. You then verify that the failure intensity of the software falls within an acceptable interval around λ by making test runs selected at random from the operational profile that you and the supplier have agreed on.

Figure 6 shows an acceptance chart for when you will accept a 10-percent chance of a false positive or a false negative that λ is in the interval $[\lambda/2, 2\lambda]$. Figure 6 is an example of how this chart might apply to the test results shown in Table 2 when λ equals 0.1 failures per CPU hour.

In this example, a total of eight CPU hours of test runs were made before the first failure. This first failure is point 1 on the chart; it falls in the continue-testing region. Similarly, if the second failure oc-

curs at 16 CPU hours, you continue to test because that is also in the continue-testing region. But if the third failure does not occur until 62 CPU hours, you are well within the acceptance region.

The final reliability-measurement task is to verify that the reliability observed in the field conforms to that observed in the system-test laboratory or validated during acceptance testing. Field reliability should improve with time — but only if the field software is corrected as failures occur.

For the static case where the field software is not corrected, you can use reported failures to compute a failure-intensity confidence interval. Again, to use reliability measurement, you must have information about the amount of software execution. One way to obtain this information is to build usage meters into the product. Such a meter might monitor transaction or command counts. Customers could then report the values of these meters periodically and whenever they reported a failure. You could then use this data to estimate total cumulative execution time.

To get statistically valid data, suppliers should devise ways to encourage customers to report *all* failures — minor as well as major. The suppliers' customer-support organizations can then categorize failures and apportion the estimated overall failure intensity to the various failure-severity classes.

Actual applications

H. Dean Drake and Duane E. Wolting of Hewlett-Packard have described the application of reliability measurement to firmware for two new computer terminals.[3] The system testing for these terminals was performed by teams of engineers, each of which was responsible for various features. The operational profile was assumed to be uniform across all the features, and the random selection of inputs from this profile was approximated by having each engineer test part-time on a haphazard, when-available basis. This ensured that the test cases were applied concurrently across all of the firmware's functional areas, and it made test coverage congruent with use coverage. Execution time was measured by keeping a log of the test periods on each test terminal. The test

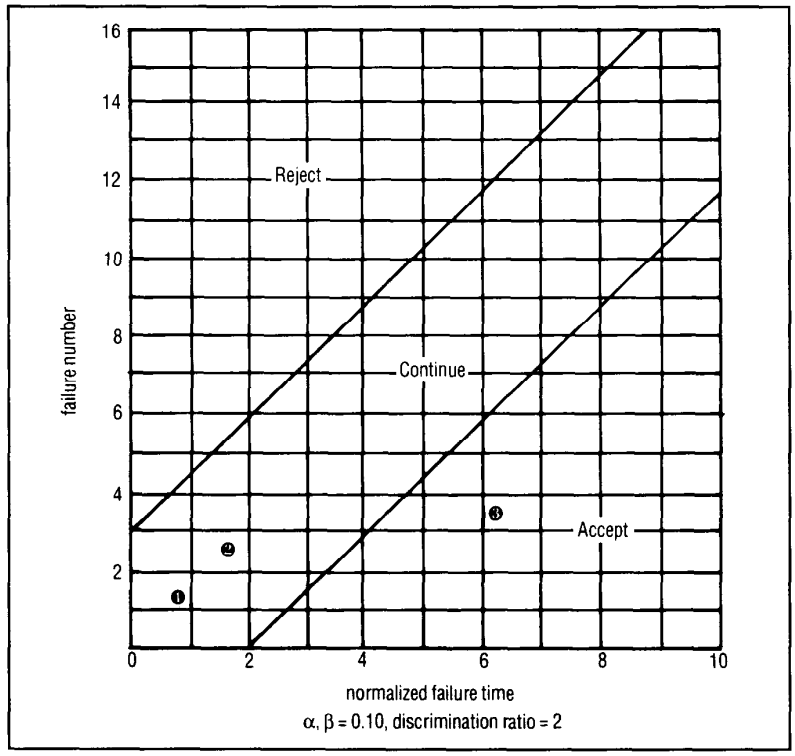

Figure 6. A reliability-measurement acceptance chart used with the failures listed in Table 2. The numbered points refer to the table's failure numbers. The discrimination ratio is the ratio by which the results can depart from the desired value.

failures discovered each day were divided by the total number of test hours for that day to determine a test-day failure intensity. These daily intensities were then averaged for successive periods of execution time. The result is the plot shown in Figure 7.

Figure 7 also shows the regression curve of the form $A \exp(Bx)$ that was fitted to this data. When x is set to the total number of test hours (647 in this case), the expression $A \exp(Bx)$ gives the estimated failure intensity at the end of testing. This value was 29.4-percent failures per test hour, so you would expect that 29.4 percent of the terminals would fail per test hour if the test were to continue with no further modifications to the software.

To be useful, you must convert this failure-intensity level to a calendar-time rate. Because the firmware is exercised whenever the customers use their terminals, the relationship between execution time and calendar time is roughly linear. Previous studies showed that customers operate this type of terminal an average of 2,000 hours out of 8,760 per year. The test-failure rate is therefore 6.71-percent failures per hour.

But this is an accelerated rate. The test engineers were exercising the terminal much harder than the average user would and were also trying to cause failures. By estimating the actual failures experienced

Table 2.
Failures used with the software-acceptance chart in Figure 6.
Times are in CPU hours.

Failure number	Failure time	Normalized failure time	Decision
1	8	0.8	Continue
2	16	1.6	Continue
3	62	6.2	λ_{obj} met

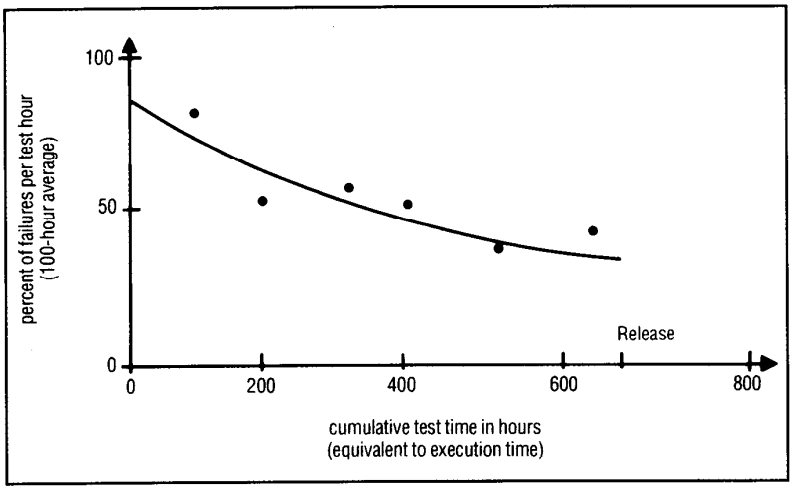

Figure 7. Actual terminal-testing failure intensities.

by this terminal in the first nine months of its distribution, you could compute an acceleration factor by dividing the predicted rate of 6.71-percent failures per hour by the observed failure rate.

The actual failure rate was estimated by assuming that the terminal firmware was repaired when a revision was installed and that a failure occurred on the same date as the repair. (The failure rate was estimated from unit revisions.) The acceleration factor estimated by this method was 47,805. When this estimate was applied to test-failure data from a second terminal development effort, the result was a predicted failure rate of 1.41-percent failures per year. The observed rate was 1.58-percent failures per year.

Dennis Christenson of AT&T has shown that reliability measurement can be used to accurately predict field failure rates in electronic switching systems.[4] The failure data in this case is the number of new problems reported by customers each day. Since electronic switching systems operate 24 hours a day and are down infrequently, the cumulative execution time is proportional to the total number of switching system modules in use each day summed over the number of days the software has been installed.

The logarithmic model is appropriate because switching-system software is updated regularly to correct the faults that underlie customer trouble reports and because the faults in the more commonly used parts of the software tend to be fixed first, so the earlier fixes tend to yield greater improvement than the later ones. When the maximum-likelihood technique was used to determine the parameters of the logarithmic Poisson model (λ_0 and θ, as described earlier), it was discovered that θ was essentially constant from release to release.

This value of θ was then used to determine λ_0 for a new release based on failures reported during 13 weeks of beta testing. With these two parameters known, it was simple to predict the total number of failures that would be observed over the full two-year life of the release. This prediction was then compared with the actual, observed number of failures. The results were that after two full years of field experience, the predicted failure intensity for release 1 was greater than the actual intensity by a maximum of 13 percent. After one year of field experience for release 2, the predicted failure intensity overestimated the actual intensity by 5 percent, Christenson reported.

The plot of the observed failures against the predicted failures is even more striking example of the degree to which a reliability model can predict failure occurrence. Figure 8 shows this plot.

One criticism often raised against the practical application of the models presented here is the assumed necessity to carefully measure execution time between failures. While the concept of relat-

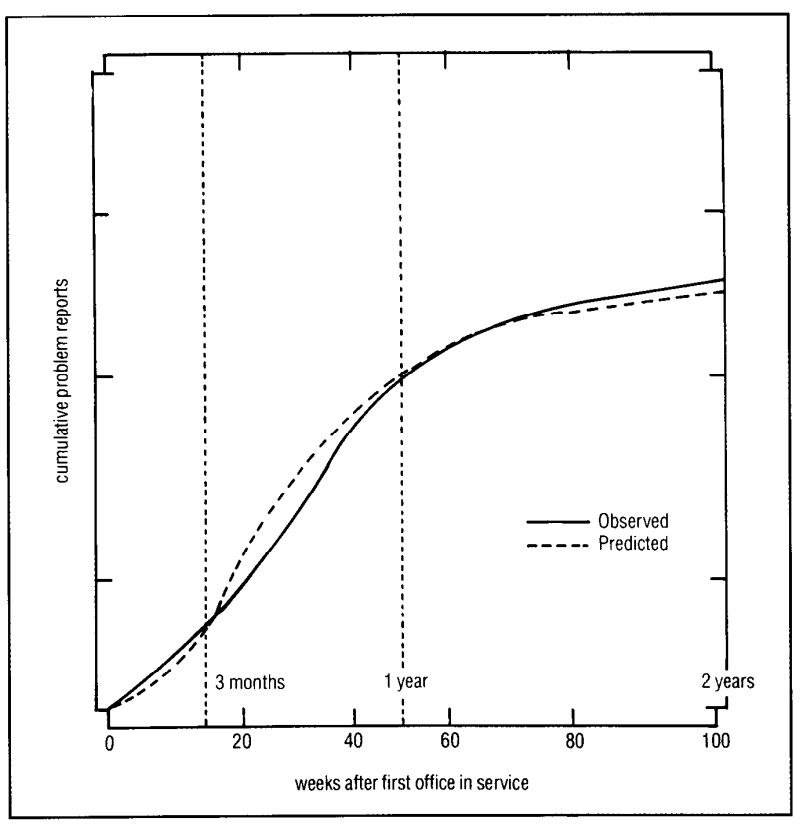

Figure 8. Predicted and observed problems for a switching office.

ing failure occurrence to execution time is fundamental, in practice you have many ways to estimate this quantity. In the Hewlett-Packard study, it was estimated from hourly terminal-test logs and the date on which each test failure occurred. In the AT&T study, execution time was estimated from total daily processor hours and the date of each unique field failure. In other applications, you might use other quantities to obtain the μ and τ observations from which you would estimate the model parameters.

How do you validate that a piece of software loaded into a processor functions correctly? The answer is that you take advantage of both the fact that you are dealing with a vast number of possible input states and the fact that for commercial-grade software only a small percentage of these states will result in failure. These conditions make a rigorous statistical approach possible — this is the essence of the technology of software-reliability measurement.

Software-reliability measurement is a new technology. There are clearly areas that need further development to make its application easier and more accurate, but it is sufficiently well developed to be used *now*. The development organizations that work on the leading edge of technology are putting it to the test and, by so doing, are adding powerful impetus to its improvement. ❖

References

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
2. ANSI/IEEE Std 830-1984, *IEEE Guide to Software-Requirements Specification*, IEEE, New York, 1984.
3. H.D. Drake and D.E. Wolting, "Reliability Theory Applied to Software Testing," *Hewlett-Packard J.*, April 1987, pp. 35-39.
4. D.A. Christenson, "Using Software-Reliability Models to Predict Field Failure Rates in Electronic Switching Systems," *Proc. Nat'l Security Industrial Assn. Ann. Joint Conf. Software Quality and Reliability*, Nat'l Security Industrial Assn., Washington, D.C., 1988.

John D. Musa is supervisor of the Software Quality Group at AT&T Bell Laboratories. He has managed or participated in many software projects. Musa has contributed extensively to the fields of software engineering and reliability measurement over the past 15 years (for which he was elected an IEEE fellow).

Musa received an MS in electrical engineering from Dartmouth College. He is a senior editor of the Software Engineering Institute book series, a founding editorial-board member of *IEEE Software*, an editor of *IEEE Proceedings*, and an editor of *Technique et Science Informatiques*.

A. Frank Ackerman is president of the Institute for Zero-Defect Software in New Providence, N.J. He has more than 25 years of experience in all phases of software development. During the past two years, he has been designing and delivering training for AT&T to support an active technology-transfer program for software-reliability measurement.

Ackerman received a PhD in computer science from the University of North Carolina at Chapel Hill and a BS in mathematics from the University of Chicago. He is a member of IEEE, ACM, and the IEEE Computer Society's Software-Engineering Standards Subcommittee.

Address questions about this article to Musa at AT&T Bell Laboratories, Rm. 6E-111B, Whippany Rd., Whippany, NJ 07981-0903.