Chapter

7

Software Reliability Measurement Experience

Allen P. Nikora
Jet Propulsion Laboratory

Michael R. Lyu
AT&T Bell Laboratories

7.1 Introduction

The key components in the SRE process, as described in Chap. 6, include reliability objective specification, operational profile determination, reliability modeling and measurement, and reliability validation. These techniques were applied to several internal projects developed within Jet Propulsion Laboratory (JPL) and Bell Communications Research (Bellcore). The project background, reliability engineering procedures, data collection efforts, modeling results, data analyses, and reliability measurements for these projects are presented in this chapter. Model comparisons for the software reliability applications, lessons learned with regard to the engineering effort, and directions for current and future software reliability investigations are also provided.

One major thing we observed is that for the failure data we analyzed, no one model was consistently the best. It was frequently the case that a model that had performed well for one set of failure data would perform badly for a different set. We therefore recommend that for any development effort, several models, each making different assumptions about the testing and debugging process, be simultaneously applied to the failure data. We also recommend that each model's applicability to the failure data be continuously monitored. Traditional goodness-of-fit tests, such as the chi-square or Kolmogorov-Smirnov tests, can be used. In addition, the model evaluation criteria described in Chap. 4 are also strongly recommended.

Another discovery is that, of the software development efforts we studied, few had quantitative reliability requirements that were measurable. Strictly speaking, it is not necessary to have a reliability requirement for a system in order to apply software reliability measurement techniques. It is quite possible to measure a software system's reliability during test and make predictions of future behavior. However, the existence of a requirement is very helpful in that:

1. Specifying a reliability requirement helps the users and developers focus on the components of the system that will have the most effect on the system's overall reliability. Potentially unreliable components can be respecified or redesigned to increase their reliability.
2. A reliability requirement will serve as a goal to be achieved during the development effort. During the testing phases, software developers and managers can estimate software reliability and determine how close it is to the required value. The difference between current and required reliability can be converted into estimates of the time and resources that will be required to achieve the goal.

We also discovered that one of the most important aspects in an SRE program is identifying the data to be collected and setting up mechanisms to ensure that the data collected are complete and accurate. We found that development organizations generally have the capability to collect the type of data that is required to use software SRE techniques. Every software development effort that we studied has a mechanism for recording and tracking failures that are observed during the testing phases and during operations. Most projects also have requirements for the test staff to keep an activity log during the testing phases. Properly used, these data collection mechanisms would provide accurate failure data in a form that could easily be used by many currently available software reliability models. However, since many software managers and developers are not aware of the types of analysis that can be done with these data, they do not devote the effort required to ensure that the collected data are complete and accurate.

Finally, we discovered that a properly defined linear combination of model results produced more accurate predictions over the set of failure data that we analyzed than any one individual model [Lyu92c]. This linear combination modeling scheme is discussed in detail.

7.2 Measurement Framework

To enhance a company's ability to deliver timely, high-quality products through an application of SRE practices, as well as to help ensure that software vendors deliver high-quality component products, several elements are included in our investigation. Figure 7.1 shows an SRE framework in our current practice. You can see that this framework is similar to that displayed in Fig. 6.1; however, it is more focused on the product life-cycle phases during system test and postdelivery.

First, customer usage is quantified by developing an operational profile. Second, quality is defined quantitatively from the customer's viewpoint by defining failures and failure severities, by determining a reliability objective, and by specifying balance among key quality objectives (e.g., reliability, delivery date, and cost) to maximize customer satisfaction. We then advocate the employment of operational profile and quality objectives to manage resources and to guide design, implementation, and testing of software. Moreover, we track reliability during testing to determine product release. This activity may be repeated until a certain reliability level has been achieved. We also analyze reliability in the field to validate the reliability engineering effort and to introduce product and process improvements.

From Fig. 7.1 we can identify four major components in the SRE process, namely, (1) *reliability objective,* (2) *operational profile,* (3) *reliability modeling and measurement,* and (4) *reliability validation.* A reliability objective is the specification of the reliability goal of a prod-

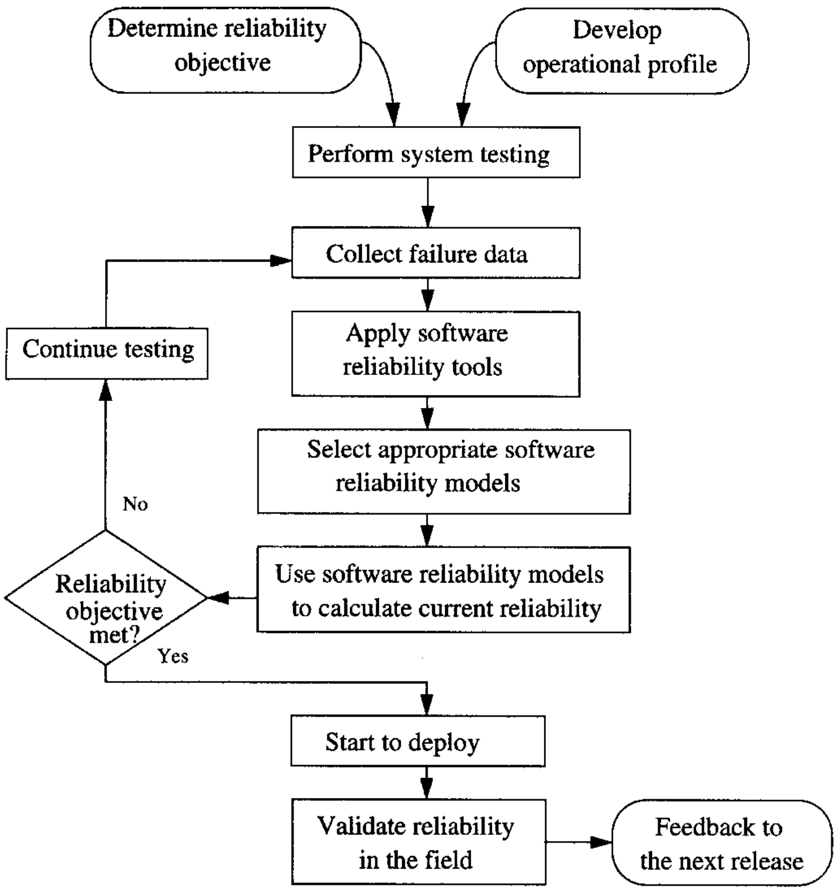

Determine reliability objective
Develop operational profile
Perform system testing
Collect failure data
Continue testing
Apply software reliability tools
Select appropriate software reliability models
No
Reliability objective met?
Use software reliability models to calculate current reliability
Yes
Start to deploy
Validate reliability in the field
Feedback to the next release

Figure 7.1 Software reliability engineering process overview.

uct from the viewpoint of the customer. If a reliability objective has been specified by the customer, that reliability objective should be used. Otherwise, you can select a reliability measure which is most intuitive and easily understood, and then determine the customer's tolerance threshold for system failures in terms of this reliability measure. For example, customer A might be mostly concerned with the total number of field failures product X may produce. Therefore, the reliability objective could be specified as, say, "product X should not produce more than 10 failures in its first 50 months of operation by customer A."

Operational profile concepts and techniques are described in Chap. 5. It is a set of disjoint alternatives of system operation and their associated probabilities of occurrence. The construction of an operational profile encourages testers to select test cases according to the system's operational usage, which contributes to more accurate estimation of software reliability in the field.

Reliability modeling is an essential element of the reliability estimation process. It determines if a product meets its reliability objective and is ready for release. You are required to use a reliability model to calculate, from failure data collected during system testing (such as failure report data and test time), various estimates of a product's reliability as a function of test time. Several interdependent estimates make equivalent statements about a product's reliability. They typically include the product's failure intensity as a function of test time t, the number of failures expected up to test time t, and the mean time to failure (MTTF) at test time t. These reliability estimates can provide the following information useful for product quality management:

1. The reliability of the product at the end of system testing.
2. The amount of (additional) test time required to reach the product's reliability objective.
3. The reliability growth as a result of testing (e.g., *failure intensity improvement factor*, defined as the ratio of the value of the failure intensity at the start of testing to the value at the end of testing).
4. The predicted reliability beyond the system testing already performed. This can be, for example, the product's reliability in the field, if the system testing has already been completed, or the predicted reliability at the end of testing, if the system testing has not yet been completed.

Chapter 3 gives a comprehensive survey on existing reliability models. Despite the existence of more than 40 models, the problem of model selection and application is manageable. Guidelines and statistical methods for selecting an appropriate model for each application are developed in Chap. 4. Furthermore, experience has shown that it is sufficient to consider only a dozen models from among the 40 models, particularly when they are already implemented in software tools (see App. A).

Using these statistical methods, best estimates of reliability are obtained during testing. These estimates are then used to project the reliability during field operation in order to determine if the reliability objective has been met. This procedure is an iterative process since more testing will be needed if the objective is not met. When the operational profile is not fully developed, application of a test compression factor can assist in estimating field reliability. A *test compression factor* is defined as the ratio of execution time required in the operational phase to execution time required in the test phase to cover the input space of the program. Since testers during testing are trying to "break" the software by searching through the input space for difficult execution conditions, while users during operation execute the software at only a normal pace, this factor represents the reduction of failure rate (or increase in reliability) during operation with respect to that observed during testing.

Finally, the projected field reliability has to be validated by comparing it with the observed field reliability. This validation not only establishes benchmarks and confidence levels of the reliability estimates, but also provides feedback to the SRE process for process improvement and better parameter tuning. For example, the model validity could be established, the growth of reliability could be determined, and the test compression factor could be refined.

Various components in this SRE framework are discussed in detail below.

7.2.1 Establishing software reliability requirements

Software reliability requirements are specified during earlier development phases, and SRE techniques are used to estimate the resources that will be required to achieve those requirements during test and operations. The resource requirements are translated into testing schedules and budgets. Resource estimates are compared to the resources actually available to make quantitative, rather than qualitative, statements concerning achievement of the reliability requirements.

7.2.1.1 Expressing software reliability. Reliability and reliability-related requirements can be expressed in one of the three following ways:

1. Probability of failure-free operation over a specified time interval
2. MTTF
3. Expected number of failures per unit time interval (failure intensity)

The first form, the basic definition of software reliability, is a probabilistic statement concerning the software's failure behavior. The other two forms can be considered relating to reliability. Reliability and reliability-related requirements must be stated in quantitative terms. Otherwise, it will not be possible to determine whether the requirements have been met. To help in understanding how to develop these requirements, examples of testable and untestable reliability requirements are given in the following paragraphs.

The following statements, paraphrased from a JPL software development effort, represent a requirement for which SRE can be used to determine the degree to which that requirement has been met.

> Reliability quantifies the ability of the system to perform a required function under the stated conditions for a period of time. Reliability is measured by the MTTF of a critical component. Under the expected operational conditions, documented elsewhere in this requirements document, the probability of the MTTF for the software being greater than or equal to 720 hours shall be 90 percent.

The above requirement is stated in a testable manner. If the expected operational conditions are stated in terms of the operational hardware configuration and the fraction of time each major functional area is expected to be used (the operational profile), the test staff can then design tests to simulate expected usage patterns and use reliability estimates made during these tests to predict operational reliability.

Confidence bounds should be associated with reliability or reliability-related requirements. If the above MTTF requirement had been stated as being simply 720 hours, it would have been possible to meet that requirement with a very wide confidence interval (for example, 90 percent probability of the MTTF lying between 200 and 1240 hours). This could have resulted in the delivery of operational software whose MTTF was considerably less than the intended 720 hours. Yet the end users of the delivered software would be told that the reliability requirement had been met. Not until the software was actually operated would the users realize the discrepancy. To avoid this problem, express the reliability requirement as the minimum value of the confidence interval. This will allow the end users to know the probability of the software meeting its reliability requirement, and permit them to plan accordingly. It is often needed to specify a tighter confidence interval. The price to pay for this improvement, though, is the need for extra validation effort to establish the tighter confidence interval.

An example of an untestable reliability-related requirement is now given. Again, the text is paraphrased from that found in a JPL development effort's system requirements document.

> The system is designed to degrade gracefully in case of failures. As a first priority, system fault protection shall ensure that no system failures or component failures will compromise system integrity. As a second priority, minimum mission science objectives previously described in this document shall not be compromised. Accordingly, each instrument shall be designed so that if one fails (either through hardware or software failures), it will not jeopardize the safety of the system or damage adjacent instruments. This includes provision for isolation from the system via the instrument power supply. If a system fault occurs, the system will automatically stop any science data gathering and go to a safe state. After a safe state is achieved and subsystems are reinitialized, science can be resumed.

The foregoing type of requirement, frequently seen in industry for critical applications, does not provide a basis for measuring the reliability of the system under development, as it contains no quantitative statements concerning the system's failure behavior. Rather, it is a statement of design constraints that are intended to localize damage resulting from a component failure to the immediate area (e.g., assembly, subsystem) in which the failure occurred. During subsequent phases of system development, it may indeed be possible to determine whether such constraints have been reflected in the system design and implementation. However, this information alone is not sufficient to make quantitative statements concerning the system's reliability. Although specifying constraints such as these is an important aspect of system specification, specific reliability requirements, similar in form to the first reliability requirement discussed in this section, would have to be provided if it were intended to use SRE techniques to determine compliance to a reliability requirement.

7.2.1.2 Specifying reliability requirements. To specify reliability requirements, use one or more of the three methods described below. The methods are [Musa87]:

1. System balance
2. Release date
3. Life-cycle cost optimization

It is possible to use one of these methods for developing the requirements for one component of the system, and another for a separate component.

The *system balance* method is primarily used to allocate reliabilities among components of a system based on the overall reliability requirement for that system. The basic principle of this method is to balance the difficulty of development work on different components of the system. The components having the most severe functional requirements or being the most technologically advanced are assigned less stringent reliability requirements. In this way, the overall reliability requirement for the system is met while minimizing the effort required to implement the most complex components. For software, this might translate to assigning less stringent reliability requirements to functions never before implemented or functions based on untried algorithms. This approach generally leads to the least costly development effort in the minimum time. The system balance method is frequently used in developing military systems.

The second approach is used when the release date is particularly critical. This is appropriate for flight systems facing a fixed launch time, or commercial systems aiming at delivery within a profit window. The release date is kept fixed in this approach. The reliability requirement is either established by available resources and funds, or is traded off against these items. With this approach, it is desirable to know how failure intensity trades off with release date. First, the way in which the failure intensity trades off with software execution time is determined. This execution time is then converted to calendar time. The following example uses the Goel-Okumoto exponential Poisson model (GO) model and Musa-Okumoto logarithmic Poisson (MO) model for illustrations.

Example 7.1 For the GO model, the relationship between the ratio of failure intensity change during test and the execution time is given by (see Sec. 3.3.2)

$$t = \frac{1}{b} \ln \frac{\lambda_0}{\lambda_F} \tag{7.1}$$

where t = elapsed execution time
λ_0 = initial failure intensity
λ_F = required failure intensity
b = failure detection rate per failure

This model also has a parameter N, which specifies the number of failures that would be observed if testing were to continue for an infinite amount of time. For the MO model, the relationship between the ratio of failure intensity change during test and the execution time is given by (see Sec. 3.5.3)

$$t = \frac{1}{\theta\lambda_0}\left(\frac{\lambda_0}{\lambda_F} - 1\right) \tag{7.2}$$

where t, λ_0, and λ_F as above
θ = failure intensity decay parameter

For this example, the failure history data from one of the testing phases of a JPL flight program (see J3 in the Data Disk) are used. Applying the GO and MO models to this data set, the following model parameter and failure intensity estimates are obtained:

Goel-Okumoto	Musa-Okumoto
λ_0 = 0.3383 failures/CPU hour	λ_0 = 0.3249 failures/CPU hour
N = 414.76 failures	θ = 0.001256/failure
b = 0.0008156 per failure	

The above equations can be used to determine the amount of test time that will be needed for various failure intensity improvement factors:

Failure intensity improvement factor	Execution time (CPU hours)	
λ_0/λ_F	GO model	MO model
10	2,823	22,052
100	5,647	242,573
1,000	8,670	2,442,782
10,000	11,293	24,499,873

Note the differences between the predictions made by the two models. In the MO model, the relationship between additional execution time needed and the improvement factor is linear, while in the GO model it is logarithmic. At this point, a choice between the two models must be made. Since it is not possible to know a priori which model is best suited to the data, the applicability of models to a set of failure data must be evaluated while the models are being applied. Once the model most applicable to the failure data has been identified, that model's relationship between failure intensity improvement factor and execution time can be used in conjunction with the relationship between execution and calendar time to determine the failure intensity requirement.

The basis of the third approach, *life-cycle cost optimization,* is the assumption that reliability improvement is obtained by more extensive testing. Costs and schedules for nontesting phases are assumed to be constant. The part of development cost due to testing decreases with higher failure intensity requirements (i.e., more failures are allowed), while the operational cost increases. The total cost therefore has a minimum. This is shown below in Fig. 7.2.

To find this minimum, testing cost as a function of failure intensity must be computed. If testing cost can be related to calendar time, and if the relationship between calendar and execution time is known [Musa87], this calculation can be done for a specific model. Similarly,

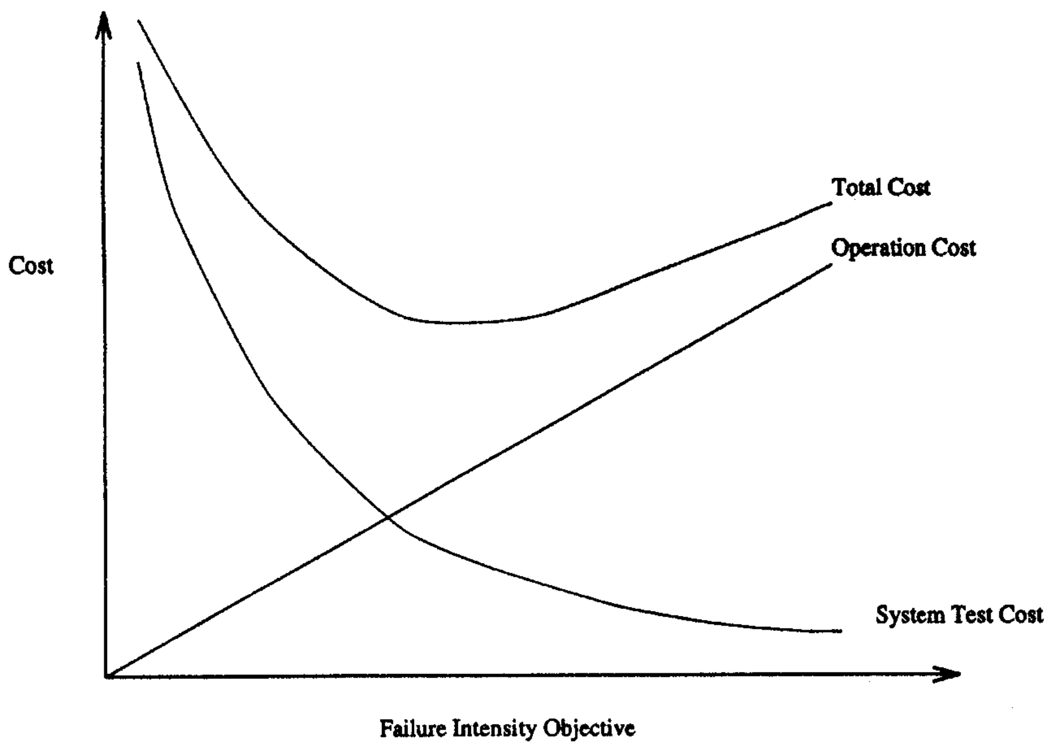


Figure 7.2 Reliability objective by life-cycle cost optimization.

the operational cost as a function of failure intensity could be computed [Ehrl93]. The following costs could be considered:

1. Terminating an improperly functioning program in an orderly manner
2. Reconstructing affected databases
3. Restarting the program
4. Determining the cause(s) of the failure
5. Developing procedures to prevent further failures of that type
6. Repairing the fault(s) causing the failure if the severity and criticality of the failure warrants corrective action
7. Testing the software to validate any repairs
8. Effect of similar failures in the future on mission or program success
9. Loss of customers' goodwill
10. Other costs, such as settling lawsuits in the event of failures in life-critical systems (e.g., commercial avionics, medical systems)

As a result, the cost of operational failures depends on many complicated factors and is hard to determine accurately. In particular, the last two types of costs may be nonlinear with increasing failure intensity.

However, we may start by assuming that failures are equally costly in each severity class. Another simple alternative is to assume all operational failures cost the same, in which case an average cost figure, normally available in each corporation, could be used. This number typically ranges from \$10,000 to \$100,000, depending on the size and the criticality of each project.

Note that the cost of testing could be expressed as an increasing function of time or number of test cases. Three conditions are normally considered: (1) the cost of testing is linear, i.e., it is constant per unit time; (2) it is linear up to time $t_{\max}$, and infinite beyond that, i.e., that project has to be finished by the time $t_{\max}$; and (3) initially it is linear, while later it increases exponentially due to the loss of credibility, lost market window, penalties, etc.

Determination of the failure intensity requirement then becomes a constrained-minimum problem that can be solved analytically or numerically. A typical economic model [Dala88, Dala90] is illustrated in Example 7.2.

Example 7.2 This model includes costs and benefits derived from trade-off between testing and operation cost due to failure. For simplicity, it is assumed that fault and failure have a one-to-one relationship, and could thus be referred simultaneously. Let

N = expected total number of faults in the program
$K(t)$ = number of faults observed up to time t
x = cost of fixing a fault when found during testing
y = cost of fixing a fault when found in the field
$c = (y - x)$ = net cost of fixing a fault after rather than before release

Further, we assume that there is a known nonnegative monotone increasing function $g(t)$ that gives the sum of the cost of testing up to time t plus the opportunity cost of not releasing the software up to time t. Here we also assume that all the failures are equally costly.

Now the following total cost of testing up to time t could be formulated as follows:

$$\begin{aligned} L(t, K(t), N) &= g(t) + xK(t) + y(N - K(t)) \\ &= g(t) - cK(t) + yN \end{aligned} \tag{7.3}$$

It can be shown that if the amount of time it takes to find a fault X during testing is distributed with a known distribution function $F_X(t)$, and the failure times are independent, then the stop-testing rule turns out to be

$$\frac{g'(t)F_X(t)}{cf_X(t)} \geq K(t) \tag{7.4}$$

where $f_X(t)$ is the density function of $F_X(t)$.

Now if we take $g(t)$ to be linear, i.e., $g(t) = g \cdot t$, and if we apply the GO model (see Sec. 3.3.2) for $F_X(t)$, namely, $F_X(t)$ is $1 - e^{-bt}$ and $f_X(t)$ is be^{-bt}, then the stopping rule in Eq. (7.4) reduces to

$$(g / bc)\,(e^{bt} - 1) \geq K(t) \tag{7.5}$$

Note that this stopping rule depends on $g(t)$ and c only through the ratio g/c. Also note that $K(t)$ can be estimated and predicted by

$$K(t) = N(1 - e^{-bt}) \tag{7.6}$$

7.2.2 Setting up a data collection process

When you set up an SRE program, you should avoid the ambition to keep every bit of information about the project and its evolvement over the life cycle. Often, people do not have a clearly defined objective for the data collection process. As a result, much effort is expended with little gain. There have been many instances in which large data collection efforts have been implemented without any capability to analyze the data. Clearly defined objectives are necessary to help define the SRE requirements. In addition, when a large amount of data are required, the development staff is usually affected. Cost and schedule suffer because of the additional effort of collecting the data. Project management complains about the large amount of overhead involved in the data collection without any constructive feedback that could help the development process.

Therefore, we recommend you use the following sequence of steps to set up a data collection process:

1. Establish the objectives. Establishing the objectives is often the distinguishing point between successful and unsuccessful data collection efforts.
2. Develop a plan for the data collection process. Involve all of the concerned parties in the data collection and analysis. This includes designers, coders, testers, quality assurance staff, and line and project software managers. This ensures that all parties understand what is being done and the impact it will have on their respective organizations. The planning should include the objectives for the data collection and a data collection plan. Address the following questions:
   a. How often will the data be gathered?
   b. By whom will the data be gathered?
   c. In what form will the data be gathered?
   d. How will the data be processed, and how will they be stored?
   e. How will the data collection process be monitored to ensure the integrity of the data and that the objectives are being met?
   f. Can existing mechanisms be used to collect the data and meet the objectives?
   g. How much effort will be required to collect the data over the life of the project?

3. If any tools have been identified in the collection process, their availability, maturity, and usability must be assessed. Commercially available tools must not be assumed to be superior to internally developed tools. Reliability, ease of use, robustness, and support are factors to be evaluated together with the application requirements. If tools are to be developed internally, plan adequate resources—cost and schedule—for the development and acceptance testing of the tool.
4. Train all parties in use of the tools. The data collectors must understand the purpose of the measurements and know explicitly what data are to be collected. Data analysts must understand a tool's analysis capabilities and limitations.
5. Perform a trial run of the data plan to iron out any problems and misconceptions. This can save a significant amount of time and effort during software development. If prototyping is being done to help specify requirements or to try out a new development method, the trial-run data collection could be done during the prototyping effort.
6. Implement the plan. Make certain that sufficient resources have been allocated to cover the required staffing and tool needs, and that the required personnel are available.
7. Monitor the process on a regular basis to provide assurance that objectives are met and that the software is meeting the established reliability goals.
8. Evaluate the data on a regular basis. Don't make the reliability assessment after software delivery. Waiting until after delivery defeats the usefulness of software reliability modeling because you have not used the information for managing the development process. Based on the experiences reported in [Lyu91a], weekly evaluation seems appropriate for many development efforts.
9. Provide feedback to all parties. This should be done as early as possible during data collection and analysis. It is especially important to do so at the end of the development effort. It is very important to provide feedback to those involved in data collection and analysis so they will be aware of the impacts of their efforts. Parties who are given feedback will be more inclined to support future efforts, as they will have a sense of efficacy and personal pride in their accomplishments.

7.2.3 Defining data to be collected

Many projects already have in place some data collection mechanisms for failure data. For example, JPL has Problem/Failure Report (P/FR), AT&T Bell Laboratories, Bellcore has Modification Request (MR) database, and IBM has Authorized Program Analysis Report (APAR). These mechanisms track the date and time at which the failure was observed, a description of the failure, and some information about the system configuration at the time the failure was observed. Specific information that needs to be collected is listed in the following subsections.

7.2.3.1 Time between successive failures. Collect the execution time between successive failures first. If execution time is unavailable, testing time between successive failures, measured by calendar time, can be used as a basis of approximation. Collect the start and completion time of each test session. If time-between-failures data cannot be collected, then collect test interval lengths and the number of failures encountered during each test interval. In many cases, this failure-count (or failure frequency) information is more easily collected than the time-between-failure information. Test interval lengths should also be accurately recorded. If possible, collect the CPU utilization during the test periods to determine the relationship between CPU and calendar time.

For many development efforts, failure-count information is the only available type. However, some software reliability tools can use only time between failures as input. In this instance, the failure-count data can be transformed to time-between-failures data in one of two ways described in Sec. 1.4. Since the uncertainty in reported failure times affects the accuracy of modeling results, problem-reporting mechanisms should be structured such that the mechanism's resolution is greater than the average interfailure time throughout the test cycle.

7.2.3.2 Functional area tested. This can be done with reference to a software requirements document or a software build plan. Reliability predictions may be dramatically different when this information is or is not available. The importance of tracking this information is illustrated in the following example.

Example 7.3 Failure data set J3 in the Data Disk is used for this example. The software reliability estimates are made using software reliability modeling tool SMERFS and CASRE (see App. A for descriptions). The Goel-Okumoto NHPP model was applied to the data. The software is assumed to be composed of two largely independent functional areas, and each functional area would be executed 50 percent of the time during operations. In producing the estimates seen in Fig. 7.3, the model was first applied to the entire set of failure data. This yields an estimated failure rate of three failures per week at week 41 of the testing phase.

The actual failure rate curve, however, is bimodal. There is clearly a change in the test procedure after week 14 of the testing phase. If the two functional areas are tested such that the first functional area is tested during the first 14 weeks,

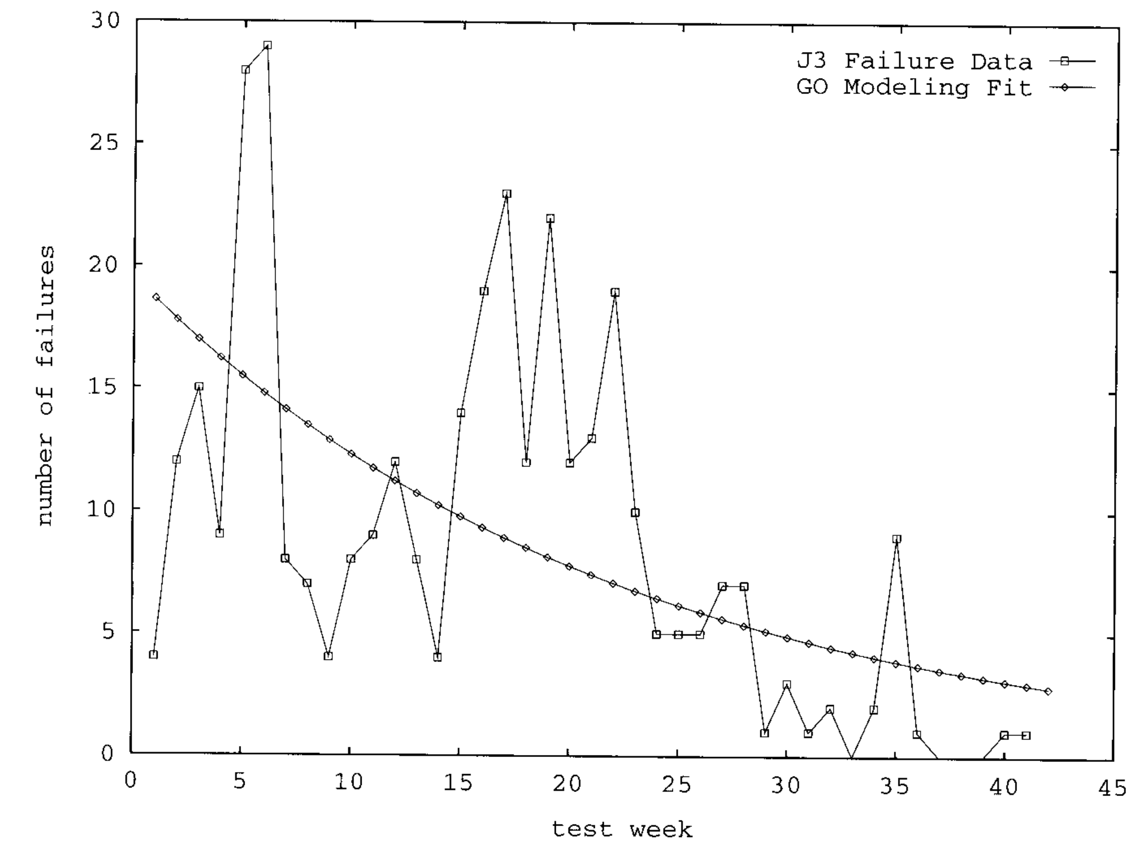


Figure 7.3 Application of GO model to entire data set.

while the second functional area is tested after week 14, then the reliabilities of the two functional areas can be separately modeled to yield a more accurate reliability estimate.

Figure 7.4 shows the reliability estimates for the two individual functional areas. By the end of week 14, the expected number of failures per week is 8 for the first functional area. During the interval between weeks 15 and 41, only the second functional area is tested. By the end of week 41, the expected number of failures per week is 1. If the software is delivered to operations at the end of week 41, it is seen that during operations, 4 failures per week can be expected while executing the first functional area, and 0.5 failures per week can be attributed to the second functional area. The resulting estimate of 4.5 failures per week is significantly different from the 3 failures per week that were estimated without taking the change in test focus into account.

As a numerical comparison, the mean square error for the predictions in Fig. 7.3 is 42.9, while that for the predictions in Fig. 7.4 is 28.3, a significant improvement. Note that the mean square error for the predictions in weeks 15–41 in Fig. 7.4 further drops to 14.2. The close fit in this period can been seen in the figure.

This analysis also shows that the first functional area needs more testing if it occurs frequently in the operational profile.

7.2.3.3 Changes during testing. Significant events that may affect the failure behavior during test include:

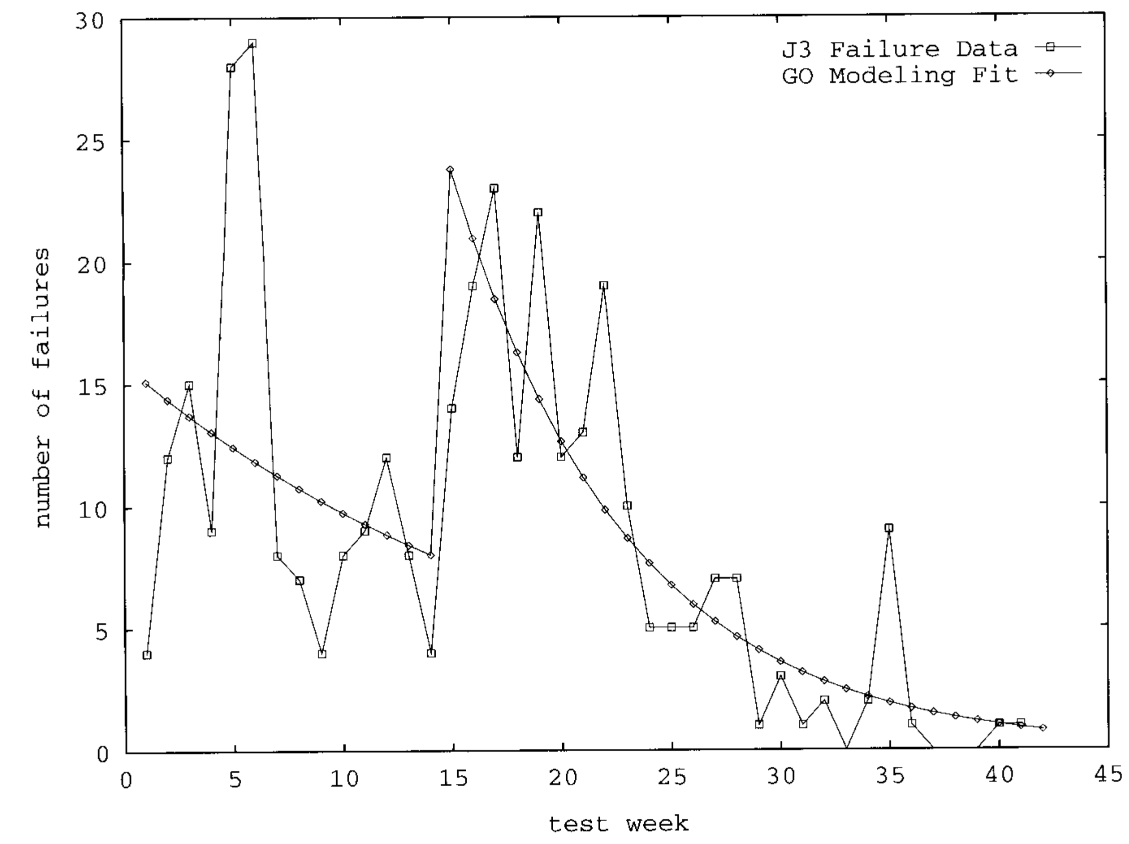


Figure 7.4 Separate GO model application of distinct functional areas.

1. *Addition of functionality to the software under test or significant modification of existing functionality.* If the software under test is still evolving, the failure intensity may be underestimated during the early stages of the program's development, yielding overly optimistic estimates of its reliability.

2. *Increases or decreases in the number of testers.* This will, for example, increase or reduce the failure-count data (expressed in calendar time) as testers are added to or taken away from the development effort. The time spent by each tester in exercising the software must be recorded so that the failure-count or time-between-failures inputs to the models are accurate.

3. *Changes in the test environment.* Examples are addition/removal of test equipment and modification of test equipment. If the test equipment is modified during a test phase to provide greater throughput, the time-between-failures and failure-count data recorded subsequently to the modification will have to be adjusted. For instance, if the clock speed in the test computer is increased by a factor of 2, the test intervals subsequent to the clock speed increase will need to be half as long as they were prior to the speedup if failure-count information is being recorded. If time-between-failures information is being recorded, the interfailure

times recorded subsequent to the speedup will have to be multiplied by 2 to be consistent with the times between failures recorded before the speedup occurred.

4. *Changes in the test method.* Examples are switching from white-box to black-box testing and changing the stress of software during test. If the test method changes during a testing effort, or if the software is exercised in a different manner, new estimates of the software's reliability will have to be made, starting at the time when the testing method or testing stress changed.

7.2.3.4 Other considerations. Interfailure times expressed in terms of CPU time are the preferred data. However, failure-count data are also recommended since existing problem-reporting mechanisms can often be used. The relative ease of collecting this information will encourage the use of SRE techniques. Currently, most problem-reporting systems collect the number of failures per unit test time interval. If your projects have existing mechanisms for collecting software failure data during developmental testing, use these data to obtain time-between-failures or failure-count data.

If failure-count data are used, a useful length for the test interval must be determined. This is influenced by such considerations as the number of testers, the number of available test sites, and the relative throughputs of test sites. Many testing teams summarize their findings on a weekly basis. Typically, a week during subsystem or system-level testing is a short enough period of time that the testing method will not change appreciably. Enough failures can be found in a week's time during the early stages of test to warrant recomputing the reliability on a weekly basis.

Many development projects require that test logs be kept during developmental and system-level testing, although the information recorded in these logs is generally not as accurate as that tracked by the problem-reporting system. Used as intended, these logs can be utilized to increase the accuracy of the failure-count or time-between-failures data available through the problem-tracking system. Without much effort beyond what is required to record failures, the following items can be collected:

1. *Functionality being tested.* The functionality can be related to items in a software build plan or requirements in a software requirements document. The reliability for each functional area should be modeled separately.
2. *Test session start date and time.*
3. *Test session stop date and time.*

In addition, it may be possible to gather CPU utilization data from the test bench's accounting facilities for each test period recorded.

If only one functional area is to be tested during a session, record only one start and stop time. If more than one functional area is to be tested, however, start and stop times should be recorded for each functional area. If testing is being done at more than one test site, keep a log at each test site. To determine test interval lengths, use the test logs from all test sites to determine the amount of testing time spent in a fixed amount of calendar time. Count the number of unique failure reports from all test sites written against that functional area in the chosen calendar interval to determine the failure-count data. These failure counts and test interval lengths can then be used as inputs to software reliability models. Note that the reliability of each functional area is separately determined.

7.2.4 Choosing a preliminary set of software reliability models

After specifying the software reliability requirements, you need to make a preliminary selection of software reliability models. Examine the assumptions that the models make about the development method and environment to determine how well they apply to your project. For instance, many models assume that the number of faults in the software has an upper bound. If software testing does not occur until the software is relatively mature and there is a low probability of making changes to the software actually being tested, models making this assumption can be included in the preliminary selection (e.g., Goel-Okumoto model, Musa basic model). If, on the other hand, significant changes are being made to the software at the same time it is being tested, it would be more appropriate to choose from those models that do not assume an upper bound to the number of faults (e.g., Musa-Okumoto and Littlewood-Verrall models). Many models also assume perfect debugging. If previous experience on similar projects indicates that most repairs do not result in new faults being inserted into the software, choose from those models making this assumption (e.g., Goel-Okumoto model, Musa-Okumoto model). However, if a significant number of repairs result in new faults being inserted into the software, it is more appropriate to choose from those models that do not assume perfect debugging (e.g., Littlewood-Verrall model).

It is important to note that there is currently no known method of evaluating these assumptions to determine a priori which model will prove optimal for a particular development effort [Abde86]. You are advised that this preliminary selection of models will be a qualitative, subjective evaluation. After a model has been selected, its performance during use can be quantitatively assessed (see Chap. 4). However, these assessment techniques cannot be applied to the preliminary selection.

If a model has been shown valid for a similar project or the early release of the same project, use that model continuously and consistently. If you have no practical experience with software reliability models, you are advised to use the following models, as recommended by AIAA [AIAA93]. The order of this list is arbitrary:

1. Generalized exponential model (see Sec. 3.7.1), which includes Jelinski-Moranda model (JM) [Jeli72] and Shooman model [Shoo73] (also in Sec. 3.3.1), Musa basic model (MB) [Musa79a] (also in Sec. 3.3.4), and Goel-Okumoto model (GO) [Goel79] (also in Sec. 3.3.2)
2. Schneidewind model (SM) [Schn75] (also in Sec. 3.3.3)
3. Musa-Okumoto logarithmic Poisson model (MO) [Musa84] (also in Sec. 3.5.3)
4. Littlewood-Verrall model (LV) [Litt73] (also in Sec. 3.6.1)

7.2.5 Choosing reliability modeling tools

Many software reliability tools are available to model and measure software reliability automatically. See App. A for a comprehensive survey.

In our study we use two tools: SRMP [Litt86] and CASRE [Lyu92d]. The CASRE tool calculates a product's reliability (the present reliability as well as future predictions of reliability) as a function of test time, and represents it in terms of several interrelated reliability measures, such as cumulative number of failures, failures per time interval, and the product's reliability function. This enables us to analyze a product's reliability from several points of view. CASRE is capable of providing product reliability estimates not only during system testing but during the product's field operation as well. In the latter case, product reliability is expressed as a function of field operation time. CASRE allows users to select and apply existing software reliability models to the data displayed in the work space. These models come from the model library of the SMERFS tool [Farr88], and consist of two categories based on their input data: time-between-failures models take the sequence of times between failures as the input, while failure-count models take number of failures per interval as the input.

7.2.6 Model application and application issues

After setting up a data collection mechanism and selecting the model(s) and tool(s), measurement of software reliability can be started. Do not attempt to measure software reliability during unit test. Although failures may be recorded during this testing phase, the individual units of code are too small to make valid software reliability estimates. Our experience indicates that the earliest point in the life cycle at which meaningful software reliability measurements can be made is at the subsystem software integration and test level. Experience gained in our study, as well as empirical evidence reported in [Musa87], suggests that software reliability measurement should not be attempted for a software system containing fewer than 2000 lines of uncommented source code. Instead, other measures (e.g., statement coverage, data and control flow coverage, data definitions and uses) could be investigated. Chapter 12 provides some emerging techniques in this area.

We now turn to the assumptions made by some of the more widely used software reliability models. Chapter 3 discusses the model assumptions in detail. These assumptions are made to cast the models into a mathematically tractable form. However, there may be situations in which the assumptions for a particular model or set of models do not apply to a development effort. In the following paragraphs, specific model assumptions are listed and the effects they may have on the accuracy of reliability estimates are described.

1. *During testing, the software is operated in a manner similar to the anticipated operational usage.* This assumption is often made to establish a relationship between the reliability behavior during testing and the operational reliability of the software. In practice, the usage pattern during testing can vary significantly from the operational usage. For instance, functionality that is not expected to be frequently used during operations (e.g., system fault protection) will be extensively tested to ensure that it functions as required when it is invoked.

When the operational usage distribution is not obtainable, one way of dealing with this issue is to model the reliability of each functional area separately, and then use the reliability of the least reliable functional area to represent the reliability of the software system as a whole. Predictions of operational reliability that are made this way will tend to be more pessimistic than the reliability that is actually observed during operations, provided that the same inputs are used during test as are used during operations. If the inputs to the software during test are different from those during operations, there will be no easily identifiable relationship between the reliability observed during test and operational reliability.

2. *There are a fixed number of faults contained in the software.* Because the mechanisms by which faults are introduced into a program during its development are poorly understood at present, this assumption is often made to make the reliability calculations more tractable. Models making this assumption should not be applied to

development efforts during which the software version being tested is simultaneously undergoing significant changes (e.g., 20 percent or more of the existing code is being changed, or the amount of code is increasing by 20 percent or more). Among the models making this assumption are the Jelinski-Moranda, the Goel-Okumoto, and the Musa Basic models. However, if the major source of change to the software during test is the correction process, and if the corrections made do not significantly change the software, it is generally safe to make this assumption. In practice, this would tend to limit application of models making this assumption to subsystem-level integration or later testing phases.

3. *No new faults are introduced into the code during the correction process.* Although introducing new faults during debugging is always possible, many models make this assumption to simplify the reliability calculations. In many development efforts, the introduction of new faults during the correction process tends to be a minor effect. If the volume of software (measured in source lines of code) being changed during correction is not a significant fraction of the volume of the entire program, and if the effects of repairs tend to be limited to the areas in which the corrections are made, this assumption is deemed acceptable. In the event that code is changing quickly or new faults are introduced when trying to fix old faults [Leve90], reliability models are still adaptable by examining the code "churns" [Dala94].

4. *Detections of failures are independent of one another.* This assumption is not necessarily valid. Indeed, evidence shows that detections of failures occur in groups, and there are some dependencies in detecting failures. However, this assumption enormously simplifies the estimation of model parameters. Determining the maximum likelihood estimator of a model parameter, for instance, requires the computation of a joint probability density function (pdf) involving all of the observed events. The assumption of independence allows this joint pdf to be computed as the product of the individual pdf's for each observation, keeping the computational requirements for parameter estimation within practical limits.

Practitioners using any currently available models have no choice but to make this assumption. Almost all the models analyzed in Chap. 3 make this assumption. Nevertheless, studies from AT&T, Hewlett Packard, and Cray Research report that the models produce fairly accurate estimates of current reliability in many situations [Ehrl90, Rapp90, Zinn90]. If the input data to the software are independent of each other, failure detection dependencies may be reduced.

When the above assumptions are deemed necessary for analytical approaches to reliability modeling, other techniques have been developed to relieve some of these assumptions. Simulation approaches (see Chap. 16) and neural networks (see Chap. 17) are two of the promising attempts.

7.2.7 Dealing with evolving software

Most models described in Chap. 3 assume that the software being tested will not be undergoing significant changes during the testing cycle. This is not always the case. A software system undergoing test may be simultaneously undergoing development, with changes being made to the existing software or new functionality being added periodically. To accurately model software reliability in this situation, changes made to the software have to be taken into account. Three approaches in handling changes to a program under test are available (see also Sec. 6.5.2.2):

1. Ignore the change.
2. Apply the component configuration change method.
3. Apply the failure time adjustment technique.

Ignoring changes is the simplest method, and is appropriate when the total volume of changes is small compared to the overall size of the program. In this case, the continual reestimation of parameters will reflect the fact that some change is in fact occurring.

The *component configuration change* approach is appropriate for the situation in which a small number of large changes are made to the software, each change resulting from the addition of independent components (e.g., addition of the telemetry gathering and downlinking capability to a spacecraft command and data subsystem). The reliability of each software component is modeled separately. The resulting estimates are then combined into a reliability figure for the overall system.

The *failure time adjustment* approach is most appropriately used when a program cannot be conveniently divided into separate independent subsystems and the program is changing rapidly enough to produce unacceptable failures in estimating the software's reliability. The three principal assumptions that are made in failure time adjustment are:

1. The program evolves sequentially. At any one time, there is only one path of evolution of the program for which reliability estimates are being made.
2. Changes in the program are due solely to growth. Differences between version k and version k+1 are due entirely to new code being added to

version k. In practice, there may be reductions in one area of the code between versions k and k+1, while growth in other areas occurs. If the reductions are small in comparison to the growth, as often occurs in repairing faults, they can usually be ignored.

3. The number of faults introduced by changes to the program are proportional to the volume of new code.

7.2.8 Practical limits in modeling ultrareliability

It is important to note a limitation of applying software reliability modeling techniques to verify systems for ultrahigh reliability (e.g., one failure per 10^7 hours of operation). It could be shown [Butl91] that quantification of software reliability in the ultrareliable regime is infeasible, since the required amount of testing time exceeds practical limits. For example, a system having a required probability of failure of 10^{-7} for a 10-hour mission implies that MTTF of the system (assuming exponentially distributed) T_F is approximately 10^8 hours. There are two basic approaches: testing with replacement, and testing without replacement. In either case, testing continues until r failures have been observed. In the first case, when a system fails, a replicated system is put on test in its place. In the second case, the failed system is not replaced. For the first case, the expected time on test, D_t, is given by

$$D_t = T_F \frac{r}{n} \tag{7.7}$$

where n is the number of items placed on test. For the second case, the expected time on test is

$$D_t = T_F \sum_{j=1}^{r} \frac{1}{n - j + 1} \tag{7.8}$$

If r is set to 1, this gives the shortest test time possible. Table 7.1 shows the expected test duration as a function of the number of test replicates, n. The expected test time with or without replacement is the same in this case.

TABLE 7.1 The Expected Test Duration as a Function of n

No. of replicates (n)	Expected test duration D_t
1	10^8 hours = 11415 years
10	10^7 hours = 1141 years
100	10^6 hours = 114 years
1,000	10^5 hours = 11.4 years
10,000	10^4 hours = 1.14 years

To get satisfactory statistical significance, larger values of r are required, which translates to even more testing. Given that economic considerations rarely allow the number of test replicates to be greater than 10, life testing of ultrareliable systems looks quite hopeless.

We should note, however, that the critical software function which requires such an ultrareliability seldom executes the complete period of the 10-hour mission. In fact, it could only require a small portion of the CPU time (e.g., the final landing approach of a long airplane flight). If we assume that only 0.1 CPU hour of the 10-hour mission requires this 10^{-7} failure probability, then T_F becomes approximately 10^6 CPU hours for the critical function. Consequently, the numbers in Table 7.1 become those in Table 7.2.

It is noted that the critical function could be tested in a more powerful CPU, which is equivalent to an increase of the number of replicates. For example, if we use 10 replicates, each running a CUP that is 100 times faster than the original one, then the goal could be practically achieved in 42 CPU days. There are, of course, obstacles to overcome before this kind of testing and validation can happen. But its achievement would not be completely infeasible.

7.3 Project Investigation at JPL

For project applications in SRE practice, we have conducted a study on SRE techniques using JPL projects. The objectives of this study include:

1. Examine the applicability of software reliability models to real-world projects.
2. Apply model selection criteria and compare models.
3. Determine if there is a best model suitable for all applications.
4. Evaluate the cost-effectiveness of SRE techniques.

7.3.1 Project selection and characterization

Data set J1. Project J1 was one of the first spacecraft in which a significant fraction of the functionality was provided by software. This software system, totaling approximately 14,000 lines of uncommented

TABLE 7.2 The Expected CPU Test Duration as a Function of n

No. of replicates (n)	Expected CPU test duration D_t
1	10^6 hours = 114 years
10	10^5 hours = 11.4 years
100	10^4 hours = 1.14 years
1,000	10^3 hours = 42 days
10,000	10^2 hours = 4.2 days

assembly language, was divided among three real-time embedded subsystems: the Attitude and Articulation Control Subsystem (AACS), the Command and Control Subsystem (CCS), and the Flight Data Subsystem (FDS). The failure data we analyzed come from spacecraft system testing, at which point the AACS, CCS, and FDS had been integrated into the spacecraft. Among the items recorded on the P/FR during system test are (1) time of failure, (2) failure type, and (3) subsystem in which the failure occurred. During J1 system test, approximately 9.5 faults per thousand lines of code (KLOC) were discovered.

Data set J2. Launched in 1989, project J2 was developed as a planetary orbiter carrying an atmospheric probe. As with the project J1, a large fraction of project J2's functionality was provided by software. Approximately 7000 uncommented source lines of HAL/S were implemented for the AACS, while about 15,000 source lines of assembly language were developed for the Command and Data Subsystem (CDS). Project J2 failure data come from spacecraft system testing. During J2 system test, approximately 10.2 faults per KLOC were discovered.

Data set J3. Failure data for the project J2 CDS during one phase of subsystem-level integration testing were available for analysis. We were able to reconstruct some elements of the testing profile. For example, it was known to us that the number of hours per week during which testing occurred was nearly constant throughout this phase, which was composed of two testing stages. In addition, the main functional areas of the software received roughly the same amount of testing every calendar week. This information resulted in the failure data being more accurate than that for other projects. During J3 subsystem test, approximately 10.1 faults per KLOC were discovered.

Data set J4. Like project J2, project J4 has an AACS and a CDS, and the number of uncommented source lines of code for each is roughly the same as that for project J2. As with projects J1 and J2, the failure data come from the spacecraft system test period. During J4 system test, approximately 8.0 faults per KLOC were discovered.

Data set J5. Project J5 is a facility for tracking and acquiring data from earth resources satellites in high-inclination orbits. Totaling about 103,000 uncommented source lines of code, the software is written in a mixture of C, Fortran, EQUEL, and OSL. About 14,000 lines were reused from previous efforts. The failure data reported here were obtained from the development organization's anomaly reporting system during software integration and test. During J5 system test, approximately 3.6 faults per KLOC were discovered.

This variety of project data would give us a chance to see whether the reliability measurement techniques developed for one type of development effort would work well for another.

7.3.2 Characterization of available data

For all of the JPL efforts, the following data were available:

1. Date on which a failure occurred
2. Failure description
3. Recommended corrective action
4. Corrective action taken
5. Date on which failure report was closed.

For each of the flight projects, the severity of each failure was also available.

Note that the following items were not systematically recorded, and were generally unavailable for use in the modeling effort:

1. *Execution times between successive failures,* or comparable information (e.g., total time spent testing during a calendar interval).
2. *Test interval lengths;* it was therefore necessary to assume that they were constant.
3. *Operational profile information* (e.g., functional area being tested, referenced to requirements or design documentation; subsystem being tested; points at which the testing method may have changed.)

The data collected from these development environments tend to be very noisy, and the assumptions of most software reliability models do not necessarily hold under the described circumstances. Nevertheless, failure data collected based on calendar time are typically under similar circumstances in many other projects.

7.3.3 Experimental results

In the reliability analysis of the JPL project data J1 through J5, we have to assume that the test time per unit interval of calendar time was relatively constant and that the testing method remained constant, since this information was not systematically recorded. Largely because of this lack of information, we decided to model the reliability of the facility as a whole, rather than attempt to model the component reliabilities. Subsequently, we applied the SRMP reliability tool to the five JPL projects and obtained the following model comparison results. Note that all the JPL project data were collected in failure-count format, and we had to convert the data into time-between-failures format for proper execution by SRMP. Random distribution of the grouped failure data was assumed for the conversion.

We evaluated a number of models surveyed in Chap. 3 and selected six models—JM, GO, MO, Duane model (DU) [Duan64] (also in Sec.

3.5.1), Littlewood model (LM) [Litt81], and LV—for project applications [Lyu91b]. Tables 7.3 to 7.7 summarize the analysis of model applicability for the JPL efforts. For each development effort, the models applied were evaluated with respect to evaluation criteria (Sec. 4.3), including model accuracy (prequential likelihood value), model bias (u-plot), bias trend (y-plot), and variability. The value for each criterion is given in the tables, while the corresponding ranking is given in parenthesis. Each of these criteria was given equal weighting in the overall ranking.

From Tables 7.3 to 7.7 we found that there was no one best model for the development efforts that were studied. This is consistent with the findings reported in Chap. 4. It is easy to see that a model that performs well for one development effort may do poorly in another. For instance, the Littlewood-Verrall model performs very well for the first three data sets—in fact, it outperforms all of the other models. However, it comes in last for the remaining two projects. This inconsistency is repeated for the other five models as well. There were no clear differences between the development processes for the five JPL applications, certainly none that would favor the selection of one model over another prior to the start of test. These findings suggest that multiple models be applied to the failure data during the test phases of a development effort, preferably models making different assumptions about the failure detection and fault removal processes. In addition, the models should be continually evaluated for applicability to the failure data. The model(s) ranking highest with respect to the evaluation criteria should then be chosen for use in predicting future reliability.

7.4 Investigation at Bellcore

After the study of JPL historical project data, we learned the lesson that data collection plays a crucial role in SRE applications. We also learned that multiple models should be applied to project data and the selection of best model(s) should be done continuously. Another study was performed to investigate Bellcore projects for SRE applications [Carm95]. The objectives of this study were:

1. Apply a better data collection effort and observe the effect.
2. Search better model(s) for particular projects as a posteriori.
3. Observe and quantify the growth of reliability during testing.
4. Classify the characteristics of reliability models.

7.4.1 Project characteristics

Project B1 is a key telecommunications software system for daily telephony operations. This system has been in existence for over 10

TABLE 7.3 **Model Rankings for J1 Data**

Measure	JM	GO	MO	DU	LM	LV
Accuracy	894.7 (6)	573.7 (3)	571.5 (2)	586.6 (4)	829.9 (5)	549.1 (1)
Bias	0.2994 (5)	0.2849 (3)	0.2849 (3)	0.2703 (2)	0.2994 (5)	0.0793 (1)
Trend	0.0995 (5)	0.0965 (3)	0.0957 (2)	0.2551 (6)	0.0994 (4)	0.0876 (1)
Variability	∞ (5)	13.81 (3)	9.225 (2)	8.402 (1)	∞ (5)	24.51 (4)
Overall rank	(6)	(3)	(2)	(4)	(5)	(1)

TABLE 7.4 **Model Rankings for J2 Data**

Measure	JM	GO	MO	DU	LM	LV
Accuracy	1074 (2)	1075 (4)	1078 (5)	1098 (6)	1074 (2)	1051 (1)
Bias	0.3378 (3)	0.3378 (3)	0.3379 (5)	0.1944 (1)	0.3382 (6)	0.2592 (2)
Trend	0.4952 (3)	0.4954 (4)	0.5041 (6)	0.4618 (2)	0.4954 (4)	0.1082 (1)
Variability	2.607 (3)	2.593 (2)	2.395 (1)	4.541 (5)	2.624 (4)	23.33 (6)
Overall rank	(2)	(3)	(5)	(4)	(6)	(1)

TABLE 7.5 **Model Rankings for J3 Data**

Measure	JM	GO	MO	DU	LM	LV
Accuracy	643.0 (3)	639.3 (2)	681.1 (5)	728.5 (6)	643.0 (3)	612.3 (1)
Bias	0.1783 (3)	0.1783 (3)	0.1700 (1)	0.1748 (2)	0.1784 (5)	−0.2581 (6)
Trend	0.3450 (3)	0.3408 (2)	0.4262 (5)	0.4282 (6)	0.3450 (3)	0.2426 (1)
Variability	4.042 (5)	3.908 (4)	2.673 (3)	2.287 (1)	4.042 (5)	2.564 (2)
Overall rank	(3)	(2)	(3)	(5)	(6)	(1)

TABLE 7.6 **Model Rankings for J4 Data**

Measure	JM	GO	MO	DU	LM	LV
Accuracy	627.1 (3)	627.1 (3)	627.1 (3)	616.0 (1)	627.1 (3)	622.9 (2)
Bias	0.2968 (4)	0.2968 (4)	0.2969 (2)	0.1858 (1)	0.2969 (2)	−0.3483 (6)
Trend	0.2399 (3)	0.2399 (3)	0.2399 (3)	0.2180 (2)	0.2399 (3)	0.1429 (1)
Variability	1.007 (1)	1.007 (1)	1.007 (1)	2.003 (5)	1.009 (4)	5.563 (6)
Overall rank	(3)	(3)	(1)	(1)	(5)	(6)

TABLE 7.7 **Model Rankings for J5 Data**

Measure	JM	GO	MO	DU	LM	LV
Accuracy	915.7 (1)	915.8 (4)	915.7 (1)	925.5 (6)	915.7 (1)	920.5 (5)
Bias	0.3023 (1)	0.3023 (1)	0.3023 (1)	0.4249 (6)	0.3023 (1)	−0.3672 (5)
Trend	0.0606 (1)	0.0615 (3)	0.0620 (4)	0.0918 (5)	0.0606 (1)	0.1009 (6)
Variability	1.587 (3)	1.540 (2)	1.395 (1)	1.650 (5)	1.589 (4)	3.189 (5)
Overall rank	(1)	(4)	(2)	(5)	(2)	(5)

years and it is deployed by all the regional Bell operating companies. The whole system includes around 1 million lines of C source code, with the main application being composed of 700K lines of code. During testing of various project B1 releases, complex interactions with other large telecommunications systems are involved. Since test metrics were collected in an automatic testing environment, some approximations to software execution time are available, which include the number and size of messages received per day, as well as total and unique test cases executed per day. The failure reports represent testing activities conducted by a group of 15 staff members that were involved with testing the current release, which includes about 250K lines of new and changed lines of code.

7.4.2 Data collection

Failure data for project B1 are collected from Bellcore internal problem tracking systems. This database stores information about all the failures (i.e., Modification Requests, or MRs) found during testing and operation. The discovery date, description, originator, severity, and other tracking information are associated with each failure. A query is available to return the number of failures found on each testing day. Staff time related information is collected manually for the project. The data collection form for this process is shown in Fig. 7.5. In addition, project B1 is able to automatically extract some time-related records to represent the intensity of testing.

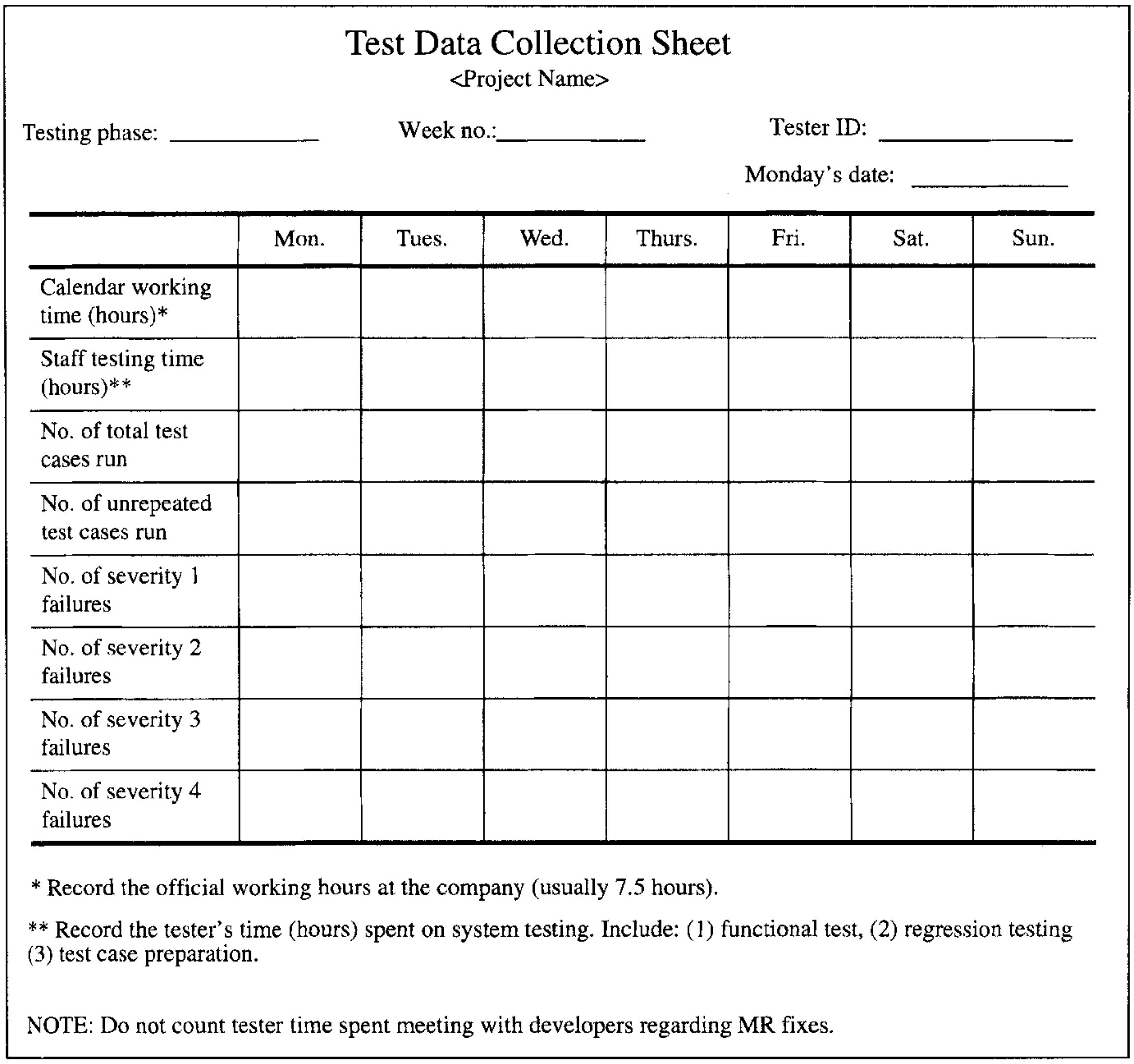
From several months' effort of software failure data collection during system testing, the following scenarios were made available for the project (B1):

1. B1.calendar: This data set collects software failure data reported during B1 system testing, based on calendar time.
2. B1.staff: This data set records B1 system testing failures based on staff time reported by each tester during testing.
3. B1.all_test: This data set records B1 failures based on the total test cases executed per testing day. The test case information is automatically collected.
4. B1.uniq_test: This data set records B1 failures based on the number of unique test cases executed each day. Repeated test cases run on the same day are not counted. This information is automatically collected.
5. B1.message: This data set records B1 failures based on the messages the system sends out. The number of messages is considered an indication of the intensity of software execution. This information is automatically collected.

7.4.3 Application results

We used the CASRE tool to apply software reliability models to the Bellcore data sets. There are two types of models in CASRE: time-between-failures (TBF) models and failure-count (FC) models. Table 7.8 presents a comparison of TBF models, including LV, geometric model (GEO) [Farr83] (also in Sec. 3.5.2), MB, JM, and MO, for the project B1 data using messages as a time measure (B1.message). Table 7.9 shows comparable results for FC models, which include generalized Poisson model (GP) [Farr83], Brooks and Motley Poisson model (BMP) and binomial model (BMB) [Farr83], NHPP (same as GO) model, and Yamada delayed S-shaped model (YSS) [Yama83] (also in Sec. 3.4.2).

The first row lists the models being compared. These are followed by the rows that record results from several model evaluation criteria, discussed in Sec. 7.3.3, and goodness-of-fit measure (Kolmogorov-Smirnov test for TBF data, chi-square test for FC data). Rank ordering



Test Data Collection Sheet

<Project Name>

Testing phase: ____________ Week no.:____________ Tester ID: ______________

Monday's date: ____________

	Mon.	Tues.	Wed.	Thurs.	Fri.	Sat.	Sun.
Calendar working time (hours)*							
Staff testing time (hours)**							
No. of total test cases run							
No. of unrepeated test cases run							
No. of severity 1 failures							
No. of severity 2 failures							
No. of severity 3 failures							
No. of severity 4 failures							

* Record the official working hours at the company (usually 7.5 hours).

** Record the tester's time (hours) spent on system testing. Include: (1) functional test, (2) regression testing (3) test case preparation.

NOTE: Do not count tester time spent meeting with developers regarding MR fixes.

Figure 7.5 Sample data collection form.

TABLE 7.8 TBF Model Comparisons for B1.message Data

Measure	LV	GEO	MB	JM	MO
Accuracy	526.8 (1)	529.5 (2)	529.8 (4)	529.5 (3)	529.9 (5)
Bias	0.127 (5)	0.113 (3)	0.080 (2)	0.067 (1)	0.120 (4)
Trend	0.0940 (1)	0.1490 (2)	0.1644 (5)	0.1643 (4)	0.1511 (3)
Variability	1.85 (2)	1.78 (1)	2.76 (4)	2.80 (5)	1.87 (3)
Goodness-of-fit	0.045 (1)	0.107 (4)	0.104 (2)	0.111 (5)	0.105 (3)
Overall rank	1	2	3	4	5

TABLE 7.9 FC Model Comparisons for B1.message Data

Measure	GP	BMP	NHPP	BMB	YSS
Accuracy	99.87 (1)	104.12 (3)	103.63 (2)	104.80 (4)	118.03 (5)
Goodness-of-fit	42.91 (1)	50.09 (2)	55.42 (4)	50.29 (3)	60.38 (5)
Degree of freedom	16	19	19	19	12
Overall rank	1	2	3	4	5

of the measure for each criterion is listed in parentheses. Overall ranks are provided in the last row. We can see for B1.message data, LV performs the best among the TBF models, while among the FC models, the GP model performs the best.

Tables 7.10 and 7.11 list, for B1.message data, the estimated times between failure, failure rates, and the factor of reliability growth for the TBF and FC models, respectively. To capture the growth of reliability from each model's viewpoint, we define a reliability growth factor (RGF) to be

$$\text{RGF} = \frac{\text{final time between failures}}{\text{initial time between failures}} \quad \text{(for TBF models)} \qquad (7.10a)$$

$$= \frac{\text{initial failure rate}}{\text{final failure rate}} \quad \text{(for FC models except the YSS model)} \qquad (7.10b)$$

TABLE 7.10 Time (Messages) Between Failures and Reliability Growth Estimated by TBF Models

Measure	GEO	JM	LV	MB	MO	Average
Initial TBF	798	896	817	909	823	849
Final TBF	3277	4213	3790	4047	3269	3719
RGF	4.11	4.70	4.64	4.45	3.97	4.37

TABLE 7.11 Failure Rates (Per Day) and Reliability Growth Estimated by FC Models

Measure	BMB	BMP	GP	NHPP	Average
Initial failure rate	3.34	3.38	2.92	3.30	3.24
Final failure rate	0.65	0.64	0.46	0.66	0.60
RGF	5.16	5.31	6.29	5.02	5.45

In other words, RGF is the same as the failure intensity improvement factor discussed in Example 7.1. Note that RGF is not defined for the YSS model.

The overall comparison of the five Bellcore data sets is summarized in Table 7.12 for the TBF models, and in Table 7.13 for the FC models. From these two tables we can see that for the Bellcore project data, LV is the best TBF model. For the FC type models, the GP model is the best one.

Table 7.14 summarizes the overall indication of level of data convergence and reliability growth for each data set, across all TBF models. Table 7.15 presents an analogous summary for the FC models. The second column in these tables, first convergence point, represents the number of data points upon which at least one model converges, divided by the total number of data points. Its percentage ratio is given in parentheses. The third column, common convergence point, is given

TABLE 7.12 Overall Model Comparisons of TBF Models

Model	GEO	JM	LV	MB	MO
B1.calendar	(5)	(3)	(1)	(2)	(4)
B1.staff	(5)	(3)	(1)	(2)	(4)
B1.all_test	(5)	(1)	(1)	(3)	(4)
B1.uniq_test	(5)	(1)	(2)	(3)	(4)
B1.message	(2)	(3)	(1)	(3)	(5)
Sum of rank	44	33	25	36	47
Total rank	(4)	(2)	(1)	(3)	(5)

TABLE 7.13 Overall Model Comparisons of FC Models

Model	BMB	BMP	GP	NHPP	YSS
B1.calendar	(4)	(2)	(3)	(1)	(4)
B1.staff	(3)	(3)	(1)	(5)	(1)
B1.all_test	(4)	(3)	(1)	(5)	(1)
B1.uniq_test	(4)	(3)	(1)	(5)	(2)
B1.message	(4)	(2)	(1)	(3)	(5)
Sum of rank	19	13	7	19	13
Total rank	(4)	(2)	(1)	(4)	(2)

by the number of points at which all models converge divided by the total number of points. The percentage ratio is also in parentheses. A high ratio indicates the parameter estimation did not converge until very late. This would happen when the data set is very noisy.

From Tables 7.14 and 7.15, we can make following observations:

1. When a reliability model is applied to a data set, it usually requires some failure observations in the estimation process for the model to get convergence. On the average, TBF models take at least 27 percent of the data to get initial convergence, while FC models require only 6 percent of the data. However, even if a model can converge very early, we still recommend that you use at least 30 observations (failures for TBF models or intervals for FC models) or 30 percent of the total data points for parameter estimation in the model application.
2. It usually takes quite a few data points before all the models converge, particularly for TBF models. In some cases a TBF model would not converge until a large amount (say, 86 percent) of the failure data are observed.
3. Not only do FC models converge earlier than TBF models, but they normally have a higher RGF than TBF models. In general, the data sets with an earlier converging point would have a larger RGF.
4. The B1 project tracks reliability growth well during testing, using either calendar time, staff time, or test-related time (test cases exe-

TABLE 7.14 Overall RGF of TBF Models for Each Data Set

Project data	First convergence point	Common convergence point	RGF
B1.calendar	31/150 (20.6%)	102/150 (68.0%)	5.10
B1.staff	34/150 (22.7%)	130/150 (86.7%)	3.57
B1.all_test	34/150 (22.7%)	130/150 (86.7%)	3.46
B1.uniq_test	77/150 (51.3%)	130/150 (86.7%)	4.20
B1.message	32/150 (21.3%)	90/150 (60.0%)	4.37
Average	(27.7%)	(77.6%)	4.14

TABLE 7.15 Overall RGF of FC Models for Each Data Set

Project data	First convergence point	Common convergence point	RGF
B1.calendar	5/85 (5.9%)	33/85 (38.8%)	6.52
B1.staff	5/85 (5.9%)	55/85 (64.7%)	4.67
B1.all_test	4/68 (5.9%)	44/68 (64.7%)	4.35
B1.uniq_test	4/68 (5.9%)	44/68 (64.7%)	5.81
B1.message	5/71 (7.0%)	14/71 (19.7%)	5.45
Average	(6.1%)	(50.5%)	5.36

cuted, messages sent by the system). This consistency indicates that this particular release has a well-controlled testing procedure and a smoothly conducted testing activity, which is confirmed when evaluating the software engineering process of the project.

7.5 Linear Combination of Model Results

Our other finding is that linear combinations of model results, even in their simplest format, appear to provide more accurate predictions than the individual models themselves [Lyu91c]. Basically, we adopt the following strategy in forming combination models:

1. Identify a basic set of models (the component models). If you can characterize the testing environment for the development effort, select models whose assumptions are closest to the actual testing practices.
2. Select models whose predictive biases tend to cancel each other. As previously described, models can have optimistic or pessimistic biases.
3. Separately apply each component model to the data.
4. Apply certain criteria to weight the selected component models (e.g., changes in the prequential likelihood) and form a combination model for the final predictions. Weights can be either static or dynamically determined.

In general, this approach is expressed as a mixed distribution,

$$\hat{f}_i(t) = \sum_{j=1}^{n} \omega_j(t)\,\hat{f}_i^j(t) \qquad (7.11)$$

where n represents the number of models, $\hat{f}_i^j(t)$ is the predictive probability density function of the jth component model, given that $i - 1$ observations of failure data have been made. Note that

$$\sum_j \omega_j(t) = 1 \qquad \text{for all } t\text{'s}$$

The linear combination model tends to preserve the features inherited from its component models. Also, because each component model performs reliability calculations independently, the combination model remains fairly simple. The component models are plugged into the combination model only at the last stage for final predictions.

Selecting appropriate component models is, of course, important to the success of the combination model. The parameter-estimation method you select to implement the component models may, to a certain extent, affect the combination model's prediction validity. We feel that the Goel-Okumoto (GO), Musa-Okumoto (MO), and Littlewood-Verrall (LV) models are the best candidates for our linear combination models. We selected them because in our investigations, we found that their predictions were valid [Lyu92c]. Other practitioners have also found that they perform well, and they are widely used [AIAA93]. Another reason is that they represent different model categories. GO represents the exponential-shaped NHPP model; MO represents the logarithmic-shaped NHPP model; and LV represents the inverse-polynomial-shaped Bayesian model. Finally, at least with the data set we analyzed, the biases of these models tend to cancel out. GO tends to be optimistic, LV tends to be pessimistic, and MO might go either way.

We experiment with three types of combinations. The goal of each is to reduce the risk of relying on a specific model, which may produce grossly inaccurate predictions, while retaining much of the simplicity of using the component models. These combinations are:

1. Statically weighted combinations
2. Dynamically weighted combinations in which weights are determined by comparing and ranking model results
3. Dynamically weighted combinations in which weights are determined by changes in model references.

7.5.1 Statically weighted linear combinations

This type of model is the simplest combination to form. Each component model has a constant weighting which remains the same throughout the modeling process. The main statically weighted combination is the equally weighted linear combination (ELC) model, which is formed by the arithmetic average of all the component models' predictions, namely,

$$\mathrm{ELC} = \frac{1}{3}\mathrm{GO} + \frac{1}{3}\mathrm{MO} + \frac{1}{3}\mathrm{LV}$$

This model follows a strategy similar to that of a Delphi survey, in which authorities working independently are asked for an opinion on a subject, and an average of the results is taken.

7.5.2 Weight determination based on ranking model results

Combination models may produce more accurate results if the weights are dynamically assigned rather than remaining static throughout the modeling process. One way of dynamically assigning weights is based on simply ranking component model results. If a combination model contains n components, choose a set of n values that can be assigned to the components based on a ranking of model results. One of the combinations is the median-weighted linear combination (MLC) model, formed by the following: for each failure, the component models would be run, and the results of the models would then be compared. The models predicting the highest and lowest times to the next failure would then be given weights of 0 in the combination, while the prediction in the middle would be given a weight of 1.

The other combination of this type is the unequally weighted linear combination (ULC) model. This model is similar to MLC except that optimistic and pessimistic predictions contribute to the final prediction. The prediction is not determined solely by the median value. Here we use weightings similar to those in the *Program Evaluation and Review Technique:*

$$\frac{1}{6}O + \frac{4}{6}M + \frac{1}{6}P$$

where O represents an optimistic prediction, P a pessimistic prediction, and M the median prediction.

7.5.3 Weight determination based on changes in prequential likelihood

The dynamically weighted linear combination (DLC) model is the one in which weights are both dynamically determined and assigned. The basis for determining and assigning weights is the changes in model preferences over a small number of observations.

In the DLC model, we assume that the applicability of any individual model to the project data may change as testing progresses. Therefore, the component models' weights will change according to changes in a model's applicability. Here, we use changes in prequential likelihood—a measure that denotes a model's accumulated accuracy—to assign weights to the component models, which could be taken over a few or many time frames. As a baseline, we formed the simplest DLC model by choosing an observation window of one time frame before each prediction as the reference in assigning weights.

7.5.4 Modeling results

In order to determine the validity of the linear combination modeling scheme, we use six models selected in Sec. 7.3.3 as a reference group to compare with the experimental group of linear combination models.

Modeling results for this comparison came from three sets of published data in [Musa79]. These data sets are also listed as SYS1, SYS2, and SYS3 in the Data Disk. Table 7.16 shows the result of SYS3 data application.

In Table 7.16, numbers in each row represent the computed measure under each criterion, with ranks in parentheses corresponding to the models in columns. The last row, "Overall rank," was determined by equally treating all the four criteria. Note that the "starting data" indicates when the model predictions began; previous data points were used for parameter estimations. This starting point was chosen such that a small but reasonable set of data points could be used for the parameter estimations.

It is observed from this table that the proposed linear combination models performed relatively well compared with the other six models. Model application to other data sets in [Musa79] also showed similar results.

7.5.5 Overall project results

Tables 7.17 and 7.18 list the performance comparisons for the three data sets from [Musa79] and the five data sets from JPL (J1 through J5). The overall comparison is done by using all four measures in Table 7.17, or by using the prequential likelihood measure (the accuracy criterion) alone in Table 7.18, since it is judged to be the most important one. In general, we consider a model as being satisfactory if and only if it is ranked 4 or better out of the 10 models for a particular project. To extend this idea, we define a *handicap* value, which is calculated by subtracting 4 (the *par* value) from the rank of a model for each data set before its ranks are summed up in the overall evaluation (or subtract 32 from the "Sum of rank" row in Tables 7.17 and 7.18). A negative handicap value represents satisfactory overall performance for the eight data sets.

We can observe several important points from these summary tables:

1. There are two sets of models under investigation here: the set of single models, and the set of combination models. In general, the set of combination models perform better than the set of single models. The acceptable models (those with a negative handicap), when considering all four measuring criteria (Table 7.17), are exactly the four linear combination models. When considering the accuracy criterion alone (Table 7.18), the three acceptable models, DLC, ELC, ULC, also belong to the combination model set. By evaluating the handicap value, we also note that the combination models usually beat the other single models with a significant margin.

TABLE 7.16 **Model Comparisons for Data Set SYS3**

Recommended Models: (1) DLC, (2) MLC, (3) ULC, (4) ELC, (4) MO

	Data Set 3 (207 data points; starting data—60)									
Measure	JM	GO	MO	DU	LM	LV	ELC	ULC	MLC	DLC
Accuracy	–811.1	–811.2	–811.1	–814.3	–811.3	–812.7	–810.8	–810.8	–811.1	–809.1
	(4)	(7)	(4)	(10)	(8)	(9)	(2)	(2)	(4)	(1)
Bias	0.0835	0.0761	0.0586	0.0994	0.0829	0.0845	0.0640	0.0594	0.0586	0.0649
	(8)	(6)	(1)	(10)	(7)	(9)	(4)	(3)	(1)	(5)
Trend	0.0623	0.0663	0.0487	0.0740	0.0602	0.0630	0.0467	0.0474	0.0480	0.0462
	(7)	(9)	(5)	(10)	(6)	(8)	(2)	(3)	(4)	(1)
Variability	5.384	5.209	4.088	2.426	6.002	3.714	4.224	4.196	4.073	3.901
	(9)	(8)	(5)	(1)	(10)	(2)	(7)	(6)	(4)	(3)
Overall rank	(6)	(8)	(4)	(9)	(9)	(6)	(4)	(3)	(2)	(1)

TABLE 7.17 Overall Model Comparisons Using All Four Criteria

	Summary of Model Ranking for Each Data by All Four Criteria									
Model	JM	GO	MO	DU	LM	LV	ELC	ULC	MLC	DLC
SYS1	(10)	(9)	(1)	(6)	(8)	(6)	(4)	(2)	(3)	(5)
SYS2	(9)	(10)	(6)	(7)	(8)	(1)	(4)	(5)	(2)	(2)
SYS3	(6)	(8)	(4)	(9)	(9)	(6)	(4)	(3)	(2)	(1)
J1	(10)	(7)	(6)	(7)	(9)	(2)	(2)	(4)	(5)	(1)
J2	(5)	(7)	(10)	(6)	(9)	(4)	(1)	(3)	(8)	(2)
J3	(8)	(6)	(6)	(8)	(10)	(1)	(1)	(1)	(4)	(5)
J4	(5)	(5)	(8)	(1)	(9)	(10)	(1)	(5)	(4)	(3)
J5	(1)	(5)	(1)	(9)	(3)	(10)	(8)	(7)	(3)	(6)
Sum of rank	54	57	42	53	65	40	25	30	31	25
Handicap	+22	+25	+10	+21	+33	+8	−7	−2	−1	−7
Total rank	(8)	(9)	(6)	(7)	(10)	(5)	(1)	(3)	(4)	(1)

TABLE 7.18 Overall Model Comparisons by the Accuracy Measure

	Summary of Model Ranking for Each Data Using the Accuracy Measure									
Model	JM	GO	MO	DU	LM	LV	ELC	ULC	MLC	DLC
SYS1	(10)	(9)	(2)	(8)	(6)	(7)	(5)	(4)	(3)	(1)
SYS2	(7)	(9)	(4)	(10)	(7)	(1)	(4)	(4)	(3)	(2)
SYS3	(4)	(7)	(4)	(10)	(8)	(9)	(2)	(2)	(4)	(1)
J1	(10)	(7)	(6)	(8)	(9)	(2)	(3)	(4)	(5)	(1)
J2	(5)	(7)	(9)	(10)	(5)	(4)	(2)	(3)	(8)	(1)
J3	(6)	(5)	(8)	(10)	(6)	(2)	(3)	(4)	(8)	(1)
J4	(6)	(6)	(6)	(2)	(6)	(5)	(3)	(4)	(6)	(1)
J5	(2)	(6)	(2)	(10)	(2)	(9)	(8)	(7)	(2)	(1)
Sum of rank	50	56	41	68	49	39	30	32	39	9
Handicap	+18	+24	+9	+36	+17	+7	−2	0	+7	−23
Total rank	(8)	(9)	(6)	(10)	(7)	(4)	(2)	(3)	(4)	(1)

2. By weighting or averaging the predictions from the three well-known component models, GO, MO, and LV, the combinational models appear to be less sensitive to potential data noise than their component models and other single models. This is reflected in the investigated data sets which include both execution-time-based data and calendar-time-based data. Moreover, when we examine all project data for the evaluation criteria, we can see that the combination models could *sometimes* outperform all their component models, but they *never* perform worse than the worst component model.

3. The DLC and ELC models perform rather consistently. Most other models seem to perform well for a few data sets but poorly for other data sets, and the fluctuation in performance is significant. By preserving good properties from the three well-known models with equal weightings, the ELC model achieves a good overall performance, as expected. On the other hand, since the DLC model is allowed to dynamically change its weightings according to the outcome of the accuracy measure, it is not surprising to see it consistently produce the best accuracy measure for almost every data set. This consistency suggests that, if you use whatever accuracy measures you deem the most important as the weighting criterion in forming the DLC model, you will get the best results.

7.5.6 Extensions and alternatives

You can extend or alter our basic approach in the following ways:

1. Extend the DLC model by increasing the size of the observation window from one time frame to N time frames. The DLC model consistently produces the best accuracy measure, but with only one observation window, it might fail to note a global measurement trend. Thus, a natural extension is to enlarge the window.
2. Try to apply models other than GO, MO, and LV as component models. If some models perform well in a particular data set, they should be the candidate component models to form a combination model.
3. Use more than three models as component models. We believe that the more component models you apply, the better the prediction. However, more computations are required, and the returns may diminish as more models are added.
4. Apply alternative weighting schemes that are based on project criteria and engineering judgments. Our approach is flexible enough that you can decide how you want to form a combination model.
5. Use the combination models themselves as component models to form another combination model.

6. As the original assumptions behind each model become lost through the layers of linear combinations, a distribution-free (nonparametric) modeling technique may emerge.

In our investigation, the most promising approach was to extend the DLC model. We considered a DLC model with a fixed N window, DLC/F, and a DLC model with a sliding N window, DLC/S. Figure 7.6 shows how the two models differ.

In the DLC/F model, the weight assignments for each model are based on changes in the accuracy measure over the last N observations. The weight assignment for each model remains fixed for the next N predictions. At the end of that time, the weights are recomputed according to the changes in accuracy over the last N observations. To compute the weight of a component model, you first determine the amount of change in component model A's accuracy measure over the last N observations. You then identify component model B, the component model whose accuracy measure changed the most. The unnormalized weight for A is simply the ratio of the change in its accuracy measure to the change in B's accuracy measure.

In the DLC/S model, you recompute the weight assignments for each model at each data point, using changes in the accuracy measure over the last N observations as the basis for determining each model's weight. To compute weights for component models, the procedure is the same as that in the DLC/F model.

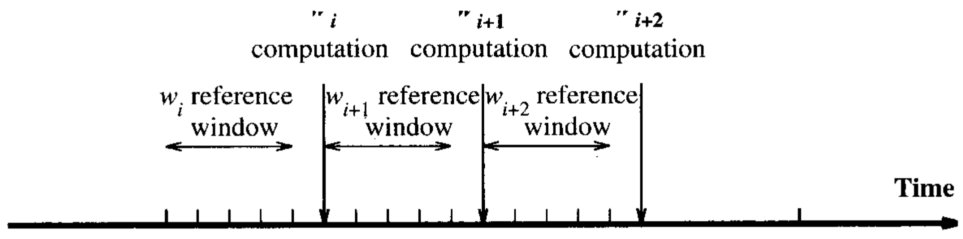


Figure 7.6a The DLC model with a fixed observation window.

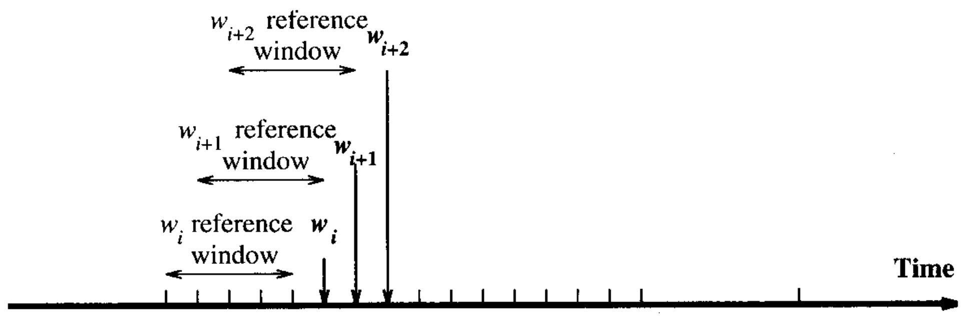


Figure 7.6b The DLC model with a sliding observation window.

Figure 7.7 summarizes the accuracy measure of the DLC/F and DLC/S models, normalized with respect to the number of measured points in each data set before being summed up for the eight data sets.

As Fig. 7.7 shows, the DLC/S model is generally superior to the DLC/F model. This result is not surprising, since DLC/S allows the observing window to advance dynamically as step-by-step prediction moves ahead. In general, the accuracy of the DLC/F model deteriorates when the window becomes larger. The DLC/S model's performance, on the other hand, improves when the window becomes larger, but only slightly larger. We found that a window size of three to four time frames is optimal.

Of course, the best window size depends on your development environment, testing scheme, and operational profile, but, in general, the window size should be fewer than five time frames, since the model is then able to catch fast shifts in model applicability among the component models.

The accuracy measure in Fig. 7.7 is the prequential likelihood, but other accuracy measures, such as the Akaike information criterion—another criterion to denote how close a prediction is to the actual data [Khos89]—or mean square error, are also feasible. The main strength of the DLC models is that they combine component models in a way that lets the output be fed back for model adjustment.

The fundamental approach of the linear combination models is simple. However, by applying more complicated procedures, we risk losing the individual model's assumptions about the physical process. It then

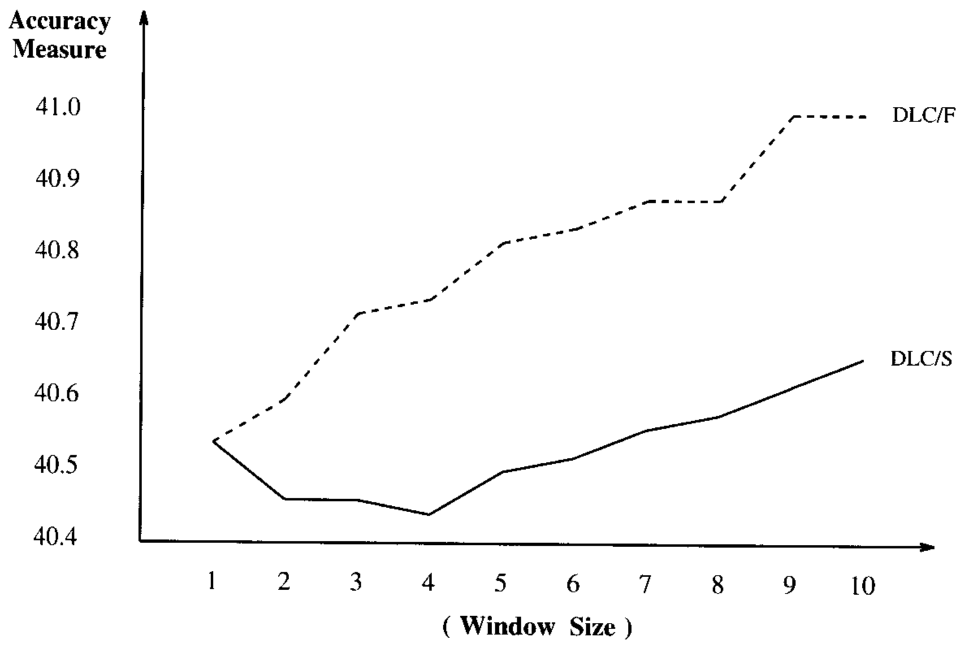


Figure 7.7 Summary of the DLC/F and DLC/S models for windows up to 10 time frames.

becomes harder to get insight into the process of reliability engineering. Most reliability models view software as a black box from which to observe failure data and make predictions. In that context, our combination models do not degrade any properties assumed in current reliability-modeling practices.

7.5.7 Long-term prediction capability

Our results have shown that the combination models perform well in making step-by-step predictions—in which you can adjust the model's parameters for each prediction—but we also want to determine how they perform in making long-term predictions, say, 20 failures ahead. For this evaluation, we select the ELC and DLC models and compare them with the GO, MO, and LV component models. Figure 7.8 shows the prediction curve for each model for the data set J3.

We use the first 152 data points J3, or up to 777 cumulative test hours as indicated by the dashed line, to estimate each model's parameters. Immediately following this estimation stage is the prediction stage. For the project J3, these two stages follow the project's natural breakdown into two testing stages.

For the DLC model, we compute model preferences and weights in the estimation phase, and fix the weight assignments in the prediction phase. From Fig. 7.8 we can observe that LV's prediction curve is too

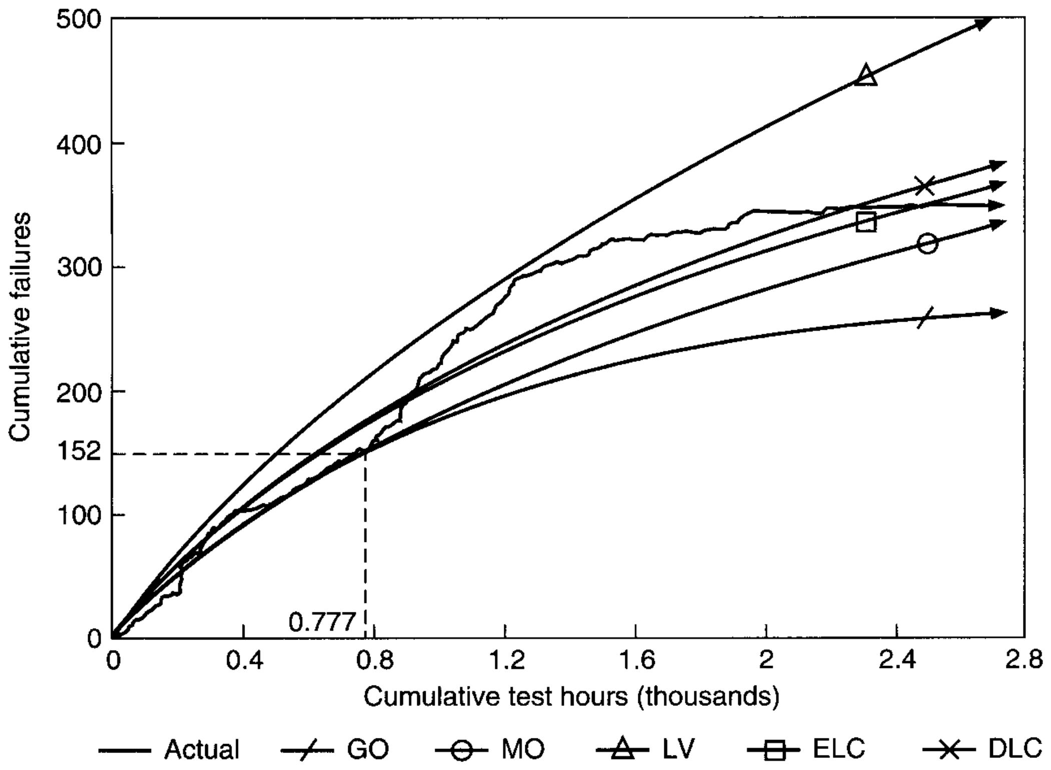


Figure 7.8 Long-term predictions for data set J3 from several models.

pessimistic, and GO's and MO's are too optimistic. In fact, all three curves for the component models are out of the actual project data curve. ELC and DLC, on the other hand, compensate these extremes and make rather reasonable long-term predictions.

To show quantitative comparisons of long-term predictions, we use mean square error instead of prequential likelihood. Prequential likelihood is more appropriate for comparing step-by-step predictions, while the mean square error provides a more widely understood measure of the distance between actual and predicted values.

Table 7.19 shows the summary of long-term predictions. The value in each project raw represents the mean square errors for a model applied to the project data in that raw, and the model ranking is shown in parentheses. The values under "Sum of mean square errors" and "Sum of ranks" indicate that the ELC and DLC models generally perform better than the component models. Even though the component models make a better prediction than the ELC and DLC models on several occasions, they also perform significantly worse on others. The ELC and DLC models, on the other hand, never make the worst long-term predictions.

7.6 Summary

We have initiated SRE programs in real-world environments for better specification and tracking of software reliability for different projects. We lay out the framework of this SRE process, which includes the most current state of practice in industry. In applying the reliability-objective-setting method and the software reliability modeling and measurement techniques, we obtain some modeling results that look promising.

TABLE 7.19 Summary of Long-Term Predictions.

Summary of Model Ranking for Long-Term Predictions Using Mean Square Errors

Data Model	GO	MO	LV	ELC	DLC
SYS1	2117 (5)	687.4 (4)	567.7 (3)	266.7 (2)	169.7 (1)
SYS2	1455 (5)	1421 (4)	246.1 (1)	930.5 (2)	955.7 (3)
SYS3	480.0 (2)	253.2 (1)	2067 (5)	745.5 (3)	779.8 (4)
J1	1089 (4)	782.9 (2)	5283 (5)	130.1 (1)	876.7 (3)
J2	4368 (4)	4370 (5)	539.3 (1)	2171 (3)	1791 (2)
J3	4712 (5)	3073 (3)	4318 (4)	1322 (2)	1141 (1)
J4	3247 (4)	3248 (5)	219.5 (1)	1684 (3)	1354 (2)
J5	60.22 (3)	60.12 (1)	104.45 (5)	68.44 (4)	60.15 (2)
Sum of mean square errors	17528.5	13896.6	13345.0	7317.3	7128.3
Sum of ranks	(32)	(25)	(25)	(20)	(18)
Overall rank	(5)	(4)	(3)	(2)	(1)

These results indicate that model performances are dramatically different depending on the context of different projects, and the use of multiple models is deemed necessary. Moreover, the reliability growth phenomenon is better demonstrated when the SRE mechanism is put in place. It is noted that the data collection process is the key to successful measurement of software reliability. In particular, failure data should be scrutinized for better classification of real software failures against the current release, and more data should be collected automatically for accuracy and reasons of economy. The application of software reliability tools also greatly simplifies the tedious job of reliability measurement and model comparison. Finally, the linear combination modeling scheme is introduced as a simple technique to increase accuracy in software reliability measurement.

In summary, we believe that when the SRE application receives more attention and wider implementation in industry, more insights leading to improvement in product quality and software development process will gradually emerge.

Problems

7.1 Analyze the failure reporting and tracking mechanisms used by a software development organization with which you're familiar. Compare the data collected by these mechanisms with the minimum set of failure data described in this chapter. Report on the suitability of your failure reporting and tracking mechanisms for software reliability estimation, and describe any changes that would result in a more suitable mechanism.

7.2 How has your development organization, or an organization you know, set reliability requirements? How do the requirements compare to the two examples given in this chapter (Sec. 7.2.1.1)? Which of the three methods of setting requirements (Sec. 7.2.1.2) has the organization used, or was another method used? Why was the method chosen?

7.3 How does your development organization, or the organization you know, estimate and forecast software reliability? How does the organization choose the model(s) used? How does the method of selection compare to that presented in this chapter?

7.4 Using the guidelines given in this chapter, write a plan that your organization could use to guide them in collecting data for software reliability measurement.

7.5 Show Eq. (7.4) is the stop-testing rule based on the assumptions in Example 7.2.

7.6 For the data set J3, suppose we have a system that is integrated in two stages. Only a portion of the system is tested during the first stage, which lasts

14 weeks. After the first stage, the remainder of the code is added to the system, at which point it is tested for 26 more weeks. At this point, the curve representing the expected number of failures will switch to the one that would have occurred for a system in its final configuration. The curve, however, is temporally translated, the amount of translation depending on the number of failures that were experienced during the first test stage.

The parameters for the first and second stages, using the Goel-Okumoto model, are as follows:

	a	*b*
First stage (weeks 1–14)	317	0.0487787
Second stage (weeks 15–40)	413	0.0461496

a. Plot the mean value function for each stage of integration, using the model parameters in the table above.
b. Determine the amount of time by which the mean value function during the second stage of integration is translated from the mean value function that would have been observed had the entire system been in place at the start of testing.

7.7 Given a project data set, the estimate of b is 1/57.6 in Eq. (7.5) for the applied GO model. The loaded salary for a single tester is \$1000 per day, and there are six full-time testers in the project. Further suppose that the cost of fixing a fault during testing is \$200, and that for fixing a fault in the field is \$2200. It is estimated that a total of 1300 faults exist in the software.
a. Draw the curves similar to Fig. 7.2. Write down the equations for each curve.
b. Draw the two curves representing the two sides of Eq. (7.5).
c. When should the testing be stopped?
d. Repeat items *b* and *c* for 10 testers and 2 testers, respectively.

7.8 Why isn't RGF defined for the YSS model?

7.9 Comparing with prequential likelihood, what are the advantages and disadvantages of using reliability growth factor (RGF) to determine the model validity during testing?

7.10 Why do combination models provide better results, on average, than individual models? Can you think of other methods to increase the prediction accuracy of models? What limitations do combination models have? Under what circumstances are these limitations important?

7.11 The linear combination models are applied mainly to one type of data in this chapter. Is it TBF data type or FC data type? For each data type, discuss which linear combination models could be formed easily, and which could not.