Chapter

11

Field Data Analysis

Wendell D. Jones
BNR Inc.

Mladen A. Vouk
North Carolina State University

11.1 Introduction

The role and functionality of software in modern computer-based systems is growing at a tremendous rate. At the same time, pressures are mounting on software developers to deliver and maintain software of better quality. Current experience indicates that, as organizations create more complex systems, software failures are an increasing proportion of system failures, while the information about these failures is frequently less than complete, uniform, or precise.

For example, field data on large telephone switching systems indicate that software is responsible for 20 to 50 percent of complete system outages. Figure 11.1 illustrates the percentage of reported causes of total system outages (due to hardware, software, and other causes) for two large telecommunications systems [Leve89, Leve90, Cram92]. The values indicated are averaged over several releases. Although both systems have similar overall functionality, there are some remarkable differences that underline an important, and often observed, property of software field reliability data—*variability.*

When examining individual releases for system A, about 30 to 60 percent of outages were attributed to hardware (some of which may have involved a combination of hardware and software problems), about 20 to 25 percent were attributed to software, while procedural and other errors accounted for the remainder of the outages [Leve90, Leve93]. In the case of system B, 3 to 7 percent of outages were attributed to hardware, and between 15 and 60 percent (depending on the maturity of the release) could be attributed to software. The figures

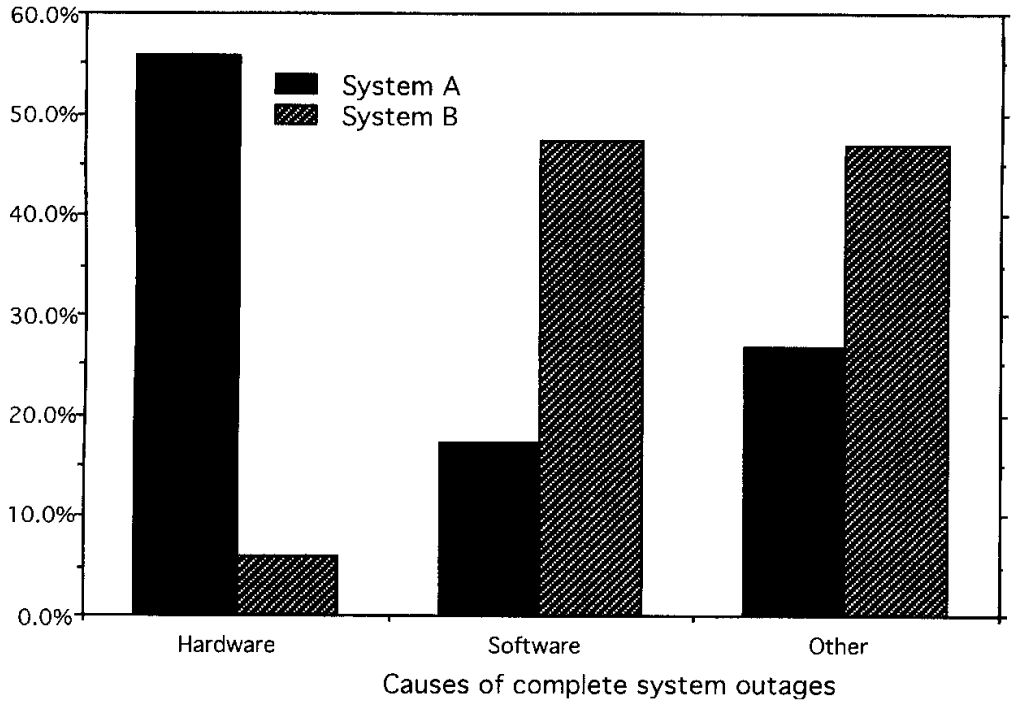

Figure 11.1 Causes of complete system outages averaged over several releases for two large telecommunication systems: System A [Leve90] and System B [Cram92].

reported for system A are closer to the distributions reported for operating systems [Iyer85a, Iyer85b]. The variance between systems A and B is due to, at least in part, the lack of a precise definition for software outage categories. It may also differ due to the functional implementation strategy of the two systems (for example, system A may implement more functionality in hardware). Whatever the reasons, it is not easy to compare the two systems and draw objective conclusions.

Examples like the one above can be found in all application areas. Therefore, it is not surprising that there are industrial, national, and international efforts to standardize software reliability data collection and analysis processes. For example, in the United States, Bellcore is an organization that acts as a software-quality watchdog from within the telecommunications community. Bellcore requires collection of outage data for all network switching elements, analysis of the data, and classification of the data by cause of failure [BELL89]. In fact, the U.S. Federal Communications Commission (FCC) mandates reporting of certain types of switch failures [FCC92].

A proper collection and analysis of *software* failure data lies at the heart of a practical evaluation of the quality of software-based systems. This is especially true when we consider analysis of software *field data* as opposed to test data. There is usually much less control over what is actually collected in the field; often analyses are based on the available historical data; and usage of the system usually cannot be stopped to

await the analysis of the data. In addition, organizations are much more sensitive to disclosure of field data due to competitive pressures.

The *goal of this chapter* is to provide insight into the process of collection and analysis of software reliability field data through a discussion of the underlying principles and case study illustrations. A distinction is made between (1) the data collected from the actual operational sites where software is used by its intended users during field tests or in day-to-day production and (2) the data collected during *controlled* system tests and experiments with operational software. The latter categories were discussed in the earlier chapters, and therefore are not considered. In the next two sections we discuss data collection principles and the basics of practical data screening and analysis. This is followed by sections that provide definition and discussion of four important topics in reliability studies—*calendar-time* analysis, *usage-time* analysis, *special-event* analysis, and *availability* analysis. Field analysis of other dependability measures, such as safety and security, is not examined in this chapter.

11.2 Data Collection Principles

11.2.1 Introduction

Software reliability is often expressed in terms of probability of failure in a given time, or in terms of the failure intensity, which is the number of failures per unit time. Minimum data requirements for calculating one expression may be slightly different than the other. Furthermore, precision in the data collection mechanism may affect the variance in reliability parameter estimates or field predictions. For example, as discussed in Chap. 1, the basic information required to perform reliability analyses includes the amount of time a software system is in operation and the exact times that failures occur. A less precise, but usable, alternative would be condensed data that reports only the total number of failures observed over a period of time. Also, additional data may be required if we wish to do more than analyze the reliability of the product. For example, if we desire to determine the availability of the product, we need both failure repair and failure severity information.

For the purpose of our discussion, we will say that whenever there is a need to make an evaluation of, or draw a conclusion based on, software reliability or availability, we conduct a *study*. Software reliability studies must have clearly defined objectives, goals, and analysis methods so that efficient use may be made of the existing data and that the cost of collecting required supplemental data is minimized. The data needed for collection and its subsequent analysis should be related to the *goals* of the study. In reliability field data analysis, some important

goals are (1) to *assess* the actual quality and reliability of a software product in its operational environment (which in turn assists in determining compliance with requirements or regulations and with the planning of maintenance resources); (2) to *relate* field failure behavior of software to its usage in the field and to its development and maintenance processes through *models;* and (3) to *predict* software behavior in the field and *control* its *field quality* by controlling its development, testing, and maintenance processes and methods.

In industry the first goal has preeminence at this time, and it is the logical first step when conducting field analysis. For example, [Hude92] provides an illustration of how field data analysis can be used to plan maintenance resources. This paper also illustrates how Nortel made progress in relation to goal 2. Goals 2 and 3 are difficult and require that field analysis be supplemented with process and product information, but achieving these goals is needed to impact the software development process and assist in its maturing. Although various organizations may have different goals, exact and detailed goals are needed to properly carry out a study [Basi84b] or any other software-related task [Boeh89, Boeh91].

11.2.2 Failures, Faults, and Related Data

Definitions for failures and faults are given in Chap. 1. More comprehensive definitions that include human errors are given in [IEEE88b, Lapr92a, Gert94]. Accurate field collection of this information and related data is essential. In addition to recording the failures and the times of corrective actions, other information is helpful for a full analysis (e.g., [IEEE88a, IEEE88b, BELL90a, Mell93]). Table 11.1 provides an example of the data that can help a designer take corrective action and also allow an analyst to properly segment and prepare data for system-level software reliability analysis. In the table, we distinguish between general classifiers, such as date and time of failure, and software-specific classifiers, such as software version information and causal analysis information.

We caution that Table 11.1 is not a form for data collection and therefore should not be used as such. The information in Table 11.1 is usually drawn from a variety of sources: customers, field support personnel, problem screeners, designers, system engineers, and maintenance personnel, including patch applicators. However, it would be very difficult for a reliability analyst to gather this information individually for all failures. Instead, what is needed is a toolset that allows integration of information (whether preexisting or current) from many sources and a variety of forms (e.g., reports, files, or databases) so that an analyst can create a table similar to Table 11.1.

TABLE 11.1 Examples of Fields Required for Reliability Analysis

Note	General classifiers	Example
Required	Date failure occurred	921214
	Time failure occurred	045600
	Date failure was reported	921214
	Tracking number or identifier (it often helps to make these identifiers as informative as possible)	ATCH-E2-1-00076
	Customer name or code	American Technology
Recommended	Site code or comparable entity	CHCGILAA34F
	Customer severity of failure (for example, critical, high, medium, low)	High
	Degradation	
	Level of degradation to system (percent)	5
	Duration of degradation (minutes)	23
	Apparent cause—top-level classification (determined at time of screening, e.g., hardware, software)	ISDN call processing
	Root cause (to be determined later by vendor)	Table control and configuration
	System or subsystem level of failure	ISDN
	Status in investigation (not under inv., under inv., closed, resolved)	Resolved
	Problem resolution process	
	Problem owner	J. Doe
	Status of problem (open, fixed, rejected, resolved)	Resolved
	Is the fix available? (Y or N)	Y
	Is this problem a duplicate of previous one? (Y or N) or the ID no. of the duplicate	N
	Resolution date	921220
	Software-specific classifiers	
Required	Software version	8.1
Recommended	Version of software in underlying operating system	4.0
	Problem at install? (Y or N)	N
	Failure type (executable or data)	Executable
	High-level cause Design or correction fault if executable Design or procedural fault if data	Design logic
	Text describing the failure or the input state that reproduces the failure	ISDN PRI trunks do not come up on warm restart when . . .
	Patch created (Y or N)	Y
	Patch process	
	Patch identifier	P-0192-064
	Status of patch (D—documented, C—coded, T—tested, A—available, GA—generally available)	GA
	Date patch is created	921221
	Version patch is written for	8.0+

A number of larger organizations have developed their own (proprietary) systems for collection and analysis of reliability data (e.g., ALCATEL, AT&T, BNR, IBM, StorageTek). There are also some commercial (e.g., [Soft93]), research (e.g., [Mos194]), and public domain [GNU95] computer-based systems for collection and analysis of software quality data (see also App. A). These systems require organization-specific customization and augmentation of their functionalities. The decisions on which data to collect, how to collect the data (for example, automated versus manual), and how to *verify correctness* of the collected information are some of the most crucial decisions an organization makes in its software reliability engineering program. Therefore they should be given appropriate attention and visibility.

Partnering with customers is essential. Without the customer's assistance it is very difficult to collect adequate field data for system analysis. The customers should know why the data are needed, how the data will be used, and how they will benefit from the analysis. Providing feedback to the customer regarding the information that is gleaned from customer field data is of great importance. It will enhance the quality of the data collected and provide customer focus that leads to quality improvement.

11.2.3 Time

In general, the more often that a (faulty) product is used, the more likely that a failure will be experienced. A full implementation of software reliability engineering requires consideration of software usage through determination of *operational profile*(s) and analysis of observed problems in that context (see Chap. 5). For example, if a software subsystem (or module) is found to exhibit an excessive number of field problems, it should be established whether this is due to very frequent usage of a component that has an average residual fault density (perhaps expressed as number of faults per line of code) or due to an excessive residual fault density in a component that is being used at the rate typical for most other product components. Reengineering of both subsystems may be required. However, the evaluation of the process that created each subsystem would be very different. For the first case, understanding the demanding requirements that are associated with the highly utilized components is of primary importance. In the second case, implementation quality is paramount. The first-case subsystem may also need extensive verification. Central to these issues is the product usage time.

Time is the execution exposure that the software receives through usage. As stated in Chap. 1, experience indicates that the best measure of time is the actual central processing unit (CPU) execution time (see also [Musa87]). However, CPU time may not be available, and it is

often possible to reformulate the measurements and *reliability models* in terms of other exposure metrics: calendar time, clock time, in-service time (usually a sum of clock times due to many software applications running simultaneously on various single- or multiple-CPU systems), logical time (such as number of executed test cases or fraction of planned test cases executed), or structural coverage (such as branch achieved statement or branch coverage) [Musa87, Tian93a, Tian93b]. In-service time usually implies that each system is treated as one unit whether it has one or several CPUs. Also, 100 months of in-service time may be associated with 50 months (clock time) of two systems or 1 month (clock time) of 100 systems. In many cases, in-service time like clock time will be proportional to system execution (CPU) time. For this chapter, the term *usage time* will refer to any of CPU, execution, clock, or in-service time.

In considering which time to use, it is necessary to weight factors such as appropriateness of the metric, availability of the data, error-sensitivity of the metric, and its relationship to a particular model. An argument in favor of using usage or calendar time instead of, for example, structural software coverage, is that engineers often are more comfortable with time than any other exposure metric. Moreover, in order to combine hardware and software reliability into one overall reliability metric, the time (whether calendar or usage) approach may be essential (see Chap. 2).

11.2.4 Usage

Ideally, one should have a record of everywhere the system is used, and some information on how it is used. An example of the needed data related to the usage information is given in Table 11.2. Also given in the table is a sample of some additional information that aids various analyses. This type of information allows calculation of metrics such as the total number of systems in operation on a given date and total operation time accumulated over all licensed systems running a particular version of the software.

Some operating systems support collection of usage data better than others. For example, processes can be created in UNIX that allow tracking of when the software is accessed, who accesses it, how frequently it is accessed, and how long the user accesses it. This allows collection of usage data at the CPU level. However, to do this in a thorough manner, more exact knowledge of the users (through licenses and other means) is often necessary, as is access to the user's system.

Although license information is often available, usage information may be less accessible. Thus, it may be necessary to statistically sample the user population to determine certain types of usage informa-

TABLE 11.2 Example of Usage Fields

Note	Field description	Example
Required	Site code or comparable entity	CHCGILFH34F
	It must match the information in the failure classification, such as the one shown in Table 11.1	
	Version of software being used	8.1
	Date software was cut over to the above version	921201
	Version of software with which the product is being used (if applicable, e.g., HP-UX 9.0)	NA
	System configuration data (e.g., hardware processor, other software loaded with the product)	MC68050
Recommended	Date/time when software installation began	921201-033001
	Date/time when software installation ended	921201-055500
	Were there any aborts? (Y or N)	N
	Is the usage under special circumstances (trials, tests, official beta testers, etc.)?	N
	Number of licenses or users at the site	1
	Last date any field in this section changed	921202

tion. In many cases, measuring the clock time associated with usage will be a sufficiently accurate measure of exposure.

For unlicensed software, or software sold through third-party vendors, the methods can be more troublesome. Crude estimates of units sold are only part of the equation, since you need to know when they were sold (and when the product is first used, or replaced with a later version or another product). You can estimate the relative usage of a product through statistical sampling methods. In any case, some combination of statistical sampling with estimates of units sold is much better than using calendar time because of the so-called loading, or ramping, effect discussed in Sec. 11.4. If this effect is not accounted for, then we are assuming constant usage over time for most models. As a consequence, field reliability may initially appear to be better than forecasts based on system testing since the system will have little usage in its early life. Later, when customers buy the new release in larger quantities and the majority of the failures occur, the reliability will be below the forecasts.

11.2.5 Data granularity

In collecting usage and other data, remember that the useful precision of the estimate/prediction of reliability is always less than the precision of the data. For example, predictions for how many failures will occur during a particular week will be of little use if the data are only

collected monthly. Therefore, choosing the right granularity is very important. For example, time intervals for data sampling or aggregation may be one second, one hour, one day, one week, one month, 10 test cases, one structural branch, or some other value. The time granularity of the raw data determines the lower limits of meaningful micromodeling and analyses that can be performed.

For a different illustration, consider prediction of the time to next failure, a standard metric in reliability analysis. With field data, predicting the time to next failure or even the next five failures is usually impractical. In many cases when field data are assimilated for analysis, groups of failures (say, 5 to 10 in size) are commonly associated with the same time frame (say, one calendar week). Predicting that the next failure will occur within the next 10 usage weeks with a probability of 0.95 will not help the customer, since 10 usage weeks may correspond to three calendar days. Thus, by the time all the data have been collected and analyzed, the next failure has *already* occurred. Field usage is very different from the laboratory test environment, where one can interrupt the testing and assess the reliability of the system before continuing with another round of tests. Field usage is continuous; therefore, analysis should be commensurate with practical data collection delays and should focus on longer-range forecasting and estimation since this can be adequately done even when the failure and/or usage data are lumped together.

11.2.6 Data maintenance and validation

In practice, a large amount of failure data may be entered manually by field support personnel from customer reports or interviews. Some software systems have internal or independent mechanisms that detect failures of various types and record that data automatically for later retrieval and processing. Even if such an automated system is in place, some data may still need to be entered manually simply because the data entry program either cannot function during a failure state or cannot recognize all failure states. Furthermore, some automated systems often cannot distinguish between hardware and software failures, and thus manual identification is required. Nevertheless, for any system, information surrounding a failure needs to be recorded as accurately as possible and data entry and database systems should be designed in such a way that all of the pertinent information is available to a reliability analyst.

Automation of date and time entries, implementation of intra- and interdata-record error and consistency checking, and standardization of entries will ensure that the analyst will have the best data possible from which to draw information. The database that holds the field data

must be updated and cross-checked as new data become available or as existing data are found to be inaccurate. The importance of consistency checking cannot be overstressed. Unfortunately, it is an area that most data collection systems overlook. The effects of data discrepancies can be very pronounced, especially in the early deployment life when the usage data are sparse. For example, even a relatively small mistake in accounting for the sites involved, or in associating failures with the appropriate software releases, can have considerable impact on the failure count and the computation of the failure intensity.

Validation and maintenance of the collected data is an absolute necessity. It is our experience that it may also be a very time-consuming and tedious activity unless appropriate tools and methods are used. If you suspect that collection errors exist, then you may need to perform an initial investigation into the amount and nature of possible data collection errors. This can be done through computations as well as visually. All information, including out-of-date records, should be kept for historical analysis purposes. The analyst's worst nightmare is coming across a database that contains all the fields required to do analysis and yet the old data is thrown away because it is no longer current and thus perceived to be of little value. For example, if there is only one record in a company's data repository for each site concerning software load information, then usually only the *current* software load information is stored and the historical usage information on past loads is not retrievable. If this occurs, then create a new database that archives records of the old usage information. It will be worth the effort after only a few months, since this information is critical for computing and comparing the field reliability of different software loads.

Distinguishing different sites or installed software systems with code identifiers is important. Codes allow for segmentation of customers and provide quicker access to the relevant information. Also, codes allow the usage to be linked with the problems experienced by customers. It is very important that the identifiers used by a customer service organization to track problems are the same as the identifiers used by the marketing/engineering personnel to track software installations, especially early in the deployment of new software. The customer information recorded by installation and shipping personnel, help-line personnel, and license agreement personnel, etc., should be consistent and available to analysts.

11.2.7 Analysis environment

For proper analysis, many pieces are required that must work well together. First, there must be processes and tools in place to collect the raw data. There must also be an appropriate storage mechanism for the

data, which is usually a database. If the database does not allow for easy data scanning, manipulation, and processing, then some other system should be in place to allow cursory examination and filtering of inappropriate or corrupt data. Of course, corrupt data should be corrected if possible, or at least marked as such. After filtering, an environment for merging data from different sources should be in place, since the data needed for failure analysis often reside in different systems. Also, some data may need to be transformed. Finally, for modeling and estimation, an environment that supports statistical methods should be available, as well as a good data-graphing tool. Depending on how the data will be used in a given environment, various information feedback mechanisms may be needed for different job roles that utilize that information.

An adequate environment may contain the following: (1) high-end networked color-graphic workstation, PC, or mainframe computing platform with a graphics terminal; (2) a data collection system linked to a multidimensional database management system; (3) manipulation, filtering, and data merging system such as SAS/BASE or SPSS; (4) software reliability modeling tools such as SMERFS, CASRE, or SoRel (see App. A); and (5) commercially available statistical analysis and visualization systems such as SAS/STAT and SAS/INSIGHT, SPSS, Systat, DataDesk, S or S+, Data Explorer, or AVS.

A general statistical analysis package is often a must, even if a reliability estimation tool is available. Most reliability packages are focused on parameter estimation and model-aptness of well-known models. However, many do not easily take into account covariates (i.e., variables that may be related to the quantity of interest, such as failure intensity, in some well-defined way), or support other standard statistical methods. Covariates are used infrequently at present but may be used more and more with the core models as analysts become more aware of their potential. Examples of covariates include a patch metric that indicates what percentage of patches are successful, and a usage-related metric that indicates how many (or what percentage of) sites have the latest software release. Both of these metrics vary with calendar or usage time and thus would be suitable covariate candidates. The patch metric may indicate local trends in the reliability growth (or degradation) if, for example, a group of patches were applied that did not fix the faults intended and instead caused additional failures. These additional failures may cause a deviation from the natural trend in reliability growth that would be explained by a patch metric covariate.

11.3 Data Analysis Principles

In statistics, analysis of data is usually considered exploratory or confirmatory. *Exploratory* analysis includes techniques in which conjec-

ture associations are only beginning, and the objective is simply to explore the potential nature of the data. *Confirmatory* techniques are typically used after some body of evidence has emerged to confirm or challenge the prevailing wisdom or current thought. The *hypothesis test* is a tool very frequently used in confirmatory analysis.

Several exploratory data analysis techniques are particularly relevant in the analysis of software failure and fault data. They are *plots and graphs, data modeling* and associated diagnostics, *data transformation,* and *data resistance.* Each technique has its own special utility, but they can often be used in combination with each other. Confirmatory analysis techniques are rarer in software reliability since the nature of software failures is still very complicated. For example, confirmatory tests in the form of *trend analysis* are described in the preceding chapter. However, there is limited agreement among researchers as to the most appropriate underlying process that is descriptive and robust enough to characterize and predict the nature of software failures. Thus exploratory techniques predominate software reliability analysis in practice.

It is often assumed that in the field software exhibits (real or apparent) reliability growth. But, this assumption needs to be validated in each study. There are two primary reasons for the assumption of reliability growth. First, most software systems can be patched relatively easily. In fact, patching is one of the great advantages of software over hardware. Faults are corrected while the system is in operation, and the system subsequently experiences reliability growth. Second, users of the system may become familiar with the imminent-failure states through firsthand experience, or information from other users or the vendor. This information tends to allow the user to avoid failure modes until a correction occurs.

In the following subsections we will examine various elementary data analysis principles. For a more complete treatment of exploratory data analysis see [Tuke77]. Visual data exploration is discussed in [Clev93, Eick92, Eick94]. We present the ideas using field data from a large release of software from a major digital telecommunications company. The data set, called DataSet 1, is on the Data Disk. In most cases, we will be concerned with reliability growth models, although most of the techniques we discuss will apply to a variety of other models and analyses.

11.3.1 Plots and graphs

Plots and graphs are very powerful tools in exploratory analysis, particularly when coupled with color graphics [Clev93]. It is often the case that an analyst can determine very quickly the initial relationships

and associations in a data set using scatterplots, line plots, stem-and-leaf plots, dot plots, and schematic or box plots. In software reliability, you often see plots of the main variables of interest. For example, for DataSet 1 the line plot in Fig. 11.2 illustrates the relationship between the total number of sites using the software release related to DataSet 1 (version N) and *calendar time*. We see that the number of offices is initially low, but quickly *ramps* up to the point of saturation. After the next release becomes available, the number of offices having version N steadily declines as customers migrate to the new release $N + 1$. This graph illustrates that the usage of version N is far from constant in calendar time, an important factor to consider when examining the reliability of this software, since usage often will not be proportional to calendar time.

Another frequently used graph in software reliability illustrates the relationship between cumulative software failures and usage time. For example, this graph is stipulated by Bellcore as a mandatory graph that U.S. telecommunications suppliers must provide in their reliability and quality reports [BELL90b]. Figure 11.3 is an example of this graph for system DataSet 1. Note that the data have been normalized to protect proprietary information. The main effect of normalization on the analysis is one of scaling. Therefore, in essence, the analysis of the nonnormalized data would be the same.

Based on Fig. 11.3, we may conjecture that some simple functional relationship may exist between cumulative failures and time. In fact, two potential functional relationships are shown in the figure using models that were defined in Chap. 3. If you think that both models

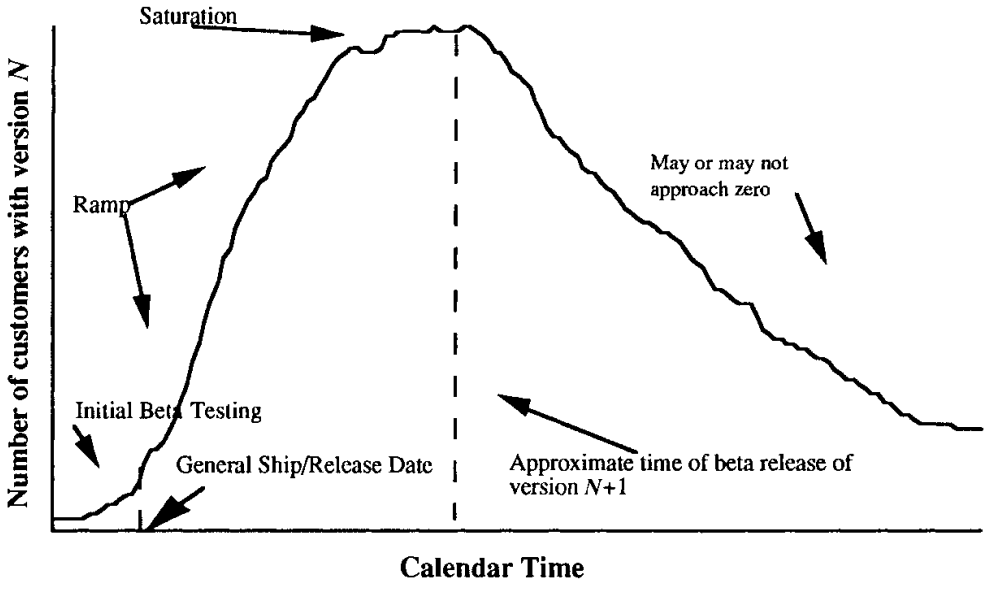

Figure 11.2 The *loading* or installation *ramping* effect (DataSet 1).

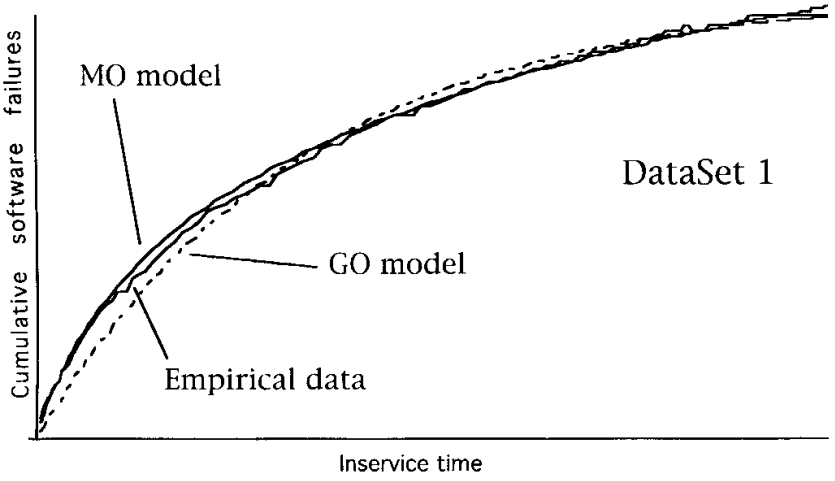

Figure 11.3 Two potential functional relationships: Musa-Okumoto (MO) model and Goel-Okumoto (GO) model for DataSet 1.

appear to fit or describe the data equally well then you have encountered the unfortunate limitations of perception with curved graphs. It is very difficult to distinguish one type of curve from another. The fitted curves are actually very different functions; one is a logarithmic function and the other is an exponential function. Therefore, the moral is *do not use* cumulative failure plots to determine functional relationships or compare different functional relationships. Although either model may be useful for interpolation, it is the extrapolation (or predictions) of the behavior that is of primary interest to a reliability engineer. These two functions have vastly different extrapolations. Thus, graphs of the cumulative failures should in practice be limited to depicting the failures for a given release against a predicted curve, or in simultaneously comparing several releases in an overlay plot.

Failure intensity is the rate of change in the expected cumulative failures. It can be quantified by the number of failures per unit time. Since the failure intensity changes over time, we are interested in the instantaneous failure intensity and how it changes with respect to time, or how it changes with accumulation of failures. Figure 11.4 is a scatterplot of the failure intensity of release DataSet 1 with a group size of 5 (percent)* against the cumulative failure count (in this case normalized to the total number of recorded failures).

Failure intensity *should* play an important role in any reliability analysis. Many of the graphs illustrated in this text (including Fig. 11.4) and many graphical diagnostics require calculation of the approximate failure intensity from empirical data. This calculation has many bene-

* Note that grouping of (unique) failures by percentage does not diminish the general analysis, even though specific values are hidden. In fact, this provides an excellent vehicle for definition of a canonical model that may carry over to the next release.

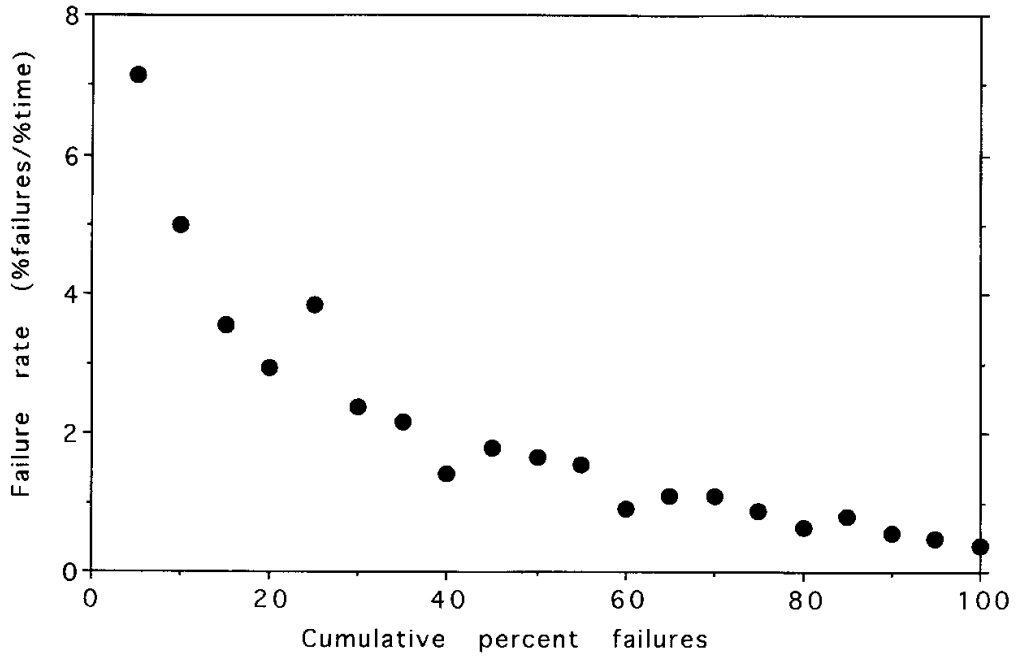

Figure 11.4 Scatterplot of the failure intensity of DataSet 1. Time is measured in in-service units.

fits: the empirical failure intensity can be measured and quantified, graphs of the failure intensity may indicate appropriate functions, and parameters for certain models may be successfully estimated from the empirical failure intensity using ordinary least squares (in addition to more complex estimation methods such as maximum likelihood). Inspection of Fig. 11.4 reveals that the failure intensity appears to decrease (indicating reliability growth) in a nonlinear fashion, and that the variance in the failure intensity becomes smaller as it approaches zero.

The obvious and uniform decreasing trend exhibited in Fig. 11.4 may not be as obvious in other situations. For example, immediately after initial deployment of a release (during the so-called transient period where the usage load is low and small errors in the data can drastically affect all metrics, including failure intensity), or where the data have large variance, we would like to confirm that reliability growth actually occurs before we commit to a particular (global) reliability growth model. The primary means for determining if reliability growth exists is the use of trend tests, which are discussed in Chap. 10. You are invited to perform the trend tests for DataSet 1 and DataSet 2 (described further in Sec. 11.5.1 and on the Data Disk) where usage is expressed in calendar-time units.

For example, when plotted, DataSet 2 exhibits unimodal failure intensity, its mean value function is S-shaped, and the reliability growth (on the calendar-time scale) does not occur prior to 8 to 10

months into the field usage.* Graphs of these quantities are given in Sec. 11.5.1. Analysis shows that one cannot be statistically certain of a confirmed global downward trend before approximately 19 months after release. This is often too long to wait for any modeling to be useful from a practical perspective. Many systems that have a unimodal field failure intensity with respect to calendar time often have a dramatically different behavior with respect to usage time (see example in Sec. 11.4). A recalibration of the data with respect to usage may greatly enhance the timeliness of the reliability modeling when usage can be estimated. In general, S-shaped models do not lend themselves to quick estimation even though the functional form may be accurate for the functional characterization considered (in this instance, failure rate versus calendar time).

An alternative may be to reduce the influence of, or even remove, the initial data where the failure intensity is increasing and fit a model on the remainder data. Examples of this approach include various forms of *data aging* such as smoothing based on a moving average (e.g., [BELL89]; also see Sec. 11.8.2) and exclusion of transient behavior data (e.g., [Mart91], see Sec. 11.5.3). The idea behind data aging is that we assign more weight to more recent failure data if we believe that the failure process changes rapidly enough that old data are not representative of the current failure process. On the other hand, if the failure process remains essentially the same throughout our observation period, then we should model using as many of the observations as possible. A discussion of this topic is given in Sec. 3.3.3; also see Sec. 11.7.2. It must be noted that weighting or elimination of early data, or weighting and elimination of outliers in general, needs to be done with great caution and with considerable expert help from a trained statistician and/or reliability engineer. It is very easy to incorrectly reduce the influence of parts of data and fit a reliability model which has no predictive validity. For example, in the case of DataSet 2, it is better to use a model which has a unimodal failure intensity function, since that behavior is characteristic of that system, than to eliminate early data.

11.3.2 Data modeling and diagnostics

Models are very important to engineers. Most useful models are predictive, and some models may be used to direct development and maintenance process management. We will assume, for convenience, that software failures occur in accordance within the general framework of the nonhomogeneous Poisson process (NHPP). In principle, this

* In fact, there appears to be an initial increase in failure intensity that may be due to the ramping effect of the user base.

assumption, which underlies many of the models, should be confirmed before we attempt to fit these models to data. A test for Poisson process assumption is described in [Cox78]. The test requires information on true interfailure times, something that may be difficult to obtain for the field data. The NHPP framework is very flexible and is not limited by specific assumptions that were common with initial models (for example, the assumption of instantaneous perfect repair of faults or the total number of failures is constant but unknown). It also allows for the use of covariates in the mean-value function that may or may not be directly tied with usage time.

11.3.3 Diagnostics for model determination

Model determination is perhaps the most underrated aspect to modeling failure data. Many naive practitioners often will examine only one model, especially when the model agrees with their assumptions or beliefs about the failure process. Also, practitioners may have viewed a graphical representation of some of the more common models, such as the GO model, and thought, *Yes, my data looks like that!* This leads to *the* question. What does failure data look like? Also, can we use the "looks" to choose the most appropriate model? The answer is a strong *yes.* There is one caveat: it depends on exactly how the data is presented.

11.3.3.1 Graphical diagnostics for model determination. Used correctly, graphical methods can be a very powerful tool to quickly determine which models may be appropriate or, especially, inappropriate. Graphical diagnostics have a long history of use in many applications, but especially in statistics. For example, there are standard plots that statisticians use to examine assumptions concerning normality of errors, or whether data from a sample is from a particular distribution. Many of these plots take advantage of the innate ability humans have in detecting straight lines; we are much less adept at distinguishing different types of curves. Many diagnostic plots graph one variable with respect to another, the object being to ascertain linearity or nonlinearity. If a linear relationship exists, then based on how the data are (or are not) transformed from the original variables, a functional relationship can be established between the two variables of interest. In our case, we are primarily interested in establishing the relationship between failures and time.

We have already seen from Fig. 11.4 that the failure intensity of release DataSet 1 is not linear with respect to the cumulative failures. Figure 11.5 shows a linear fit overlaid on the failure intensity curve. Obviously there are persistent deviations from the linear fit which would be confirmed by examining the graph of the *residuals* of the fit.

Examination of residuals *should* be a part of any serious modeling study. In our case, the residuals are serially correlated (a common symptom of a poor predictive model), indicating that the failure rate is significantly nonlinear with respect to the cumulative number of failures.

[Musa87] and [Jone91] examined the long-term relative predictive error of several models over time for many releases of field reliability data. This method is a historical validation technique and thus requires historical data if it is to be used. That is, one calculates total failure forecasts at several points earlier in the release cycle given the data that would have been available at that time. The long-term forecasts are then compared against the actual values and a relative error (in this case, a relative *predictive* error) is computed and plotted.

Consider the relative error plot for the MO model in Fig. 11.6 using the reliability data from three software releases of the same software system. A good model tends to have values of relative error close to zero, and tending toward zero over time. The variance in the relative predictive error is usually greatest at first, and then narrows as the model has both more data for model stability and as forecasts are not as far into the future. A lack of symmetry in the relative error indicates a possible bias in the candidate model and usually implies that the candidate model is either not appropriate for prediction or may need recalibration. In Fig. 11.6 we see that the MO model has little bias except at the very end where the relative error is small.

As introduced in Chap. 4, u-plots and plots based upon the prequential likelihood [Abde86, Broc92, DeGr86] are another source of candidates for graphical diagnostics. However, the mathematics are very

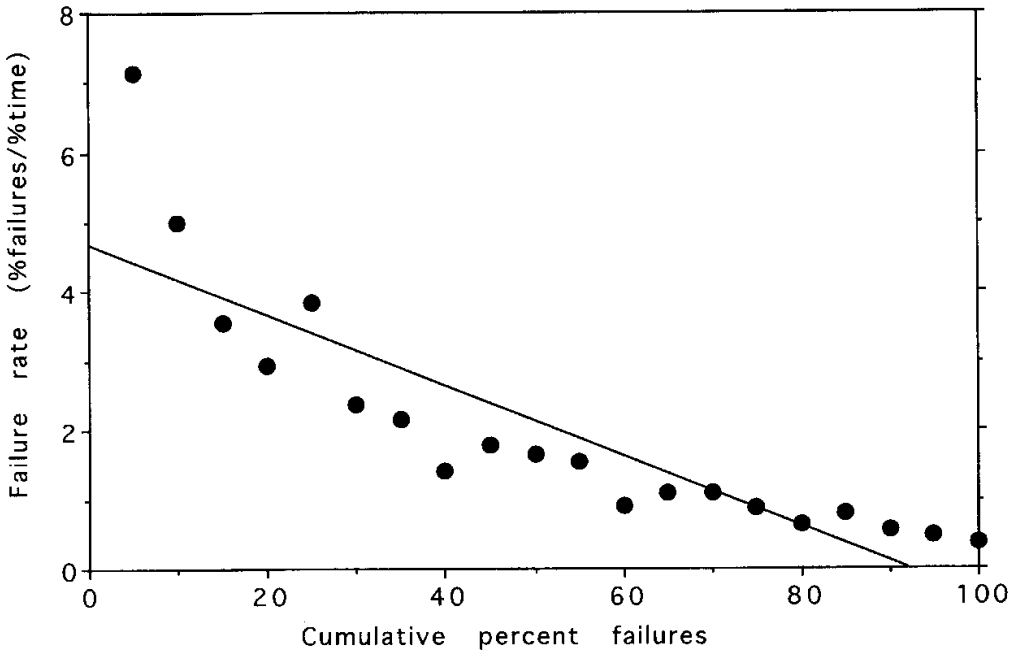

Figure 11.5 Linear fit through the failure intensity of DataSet 1.

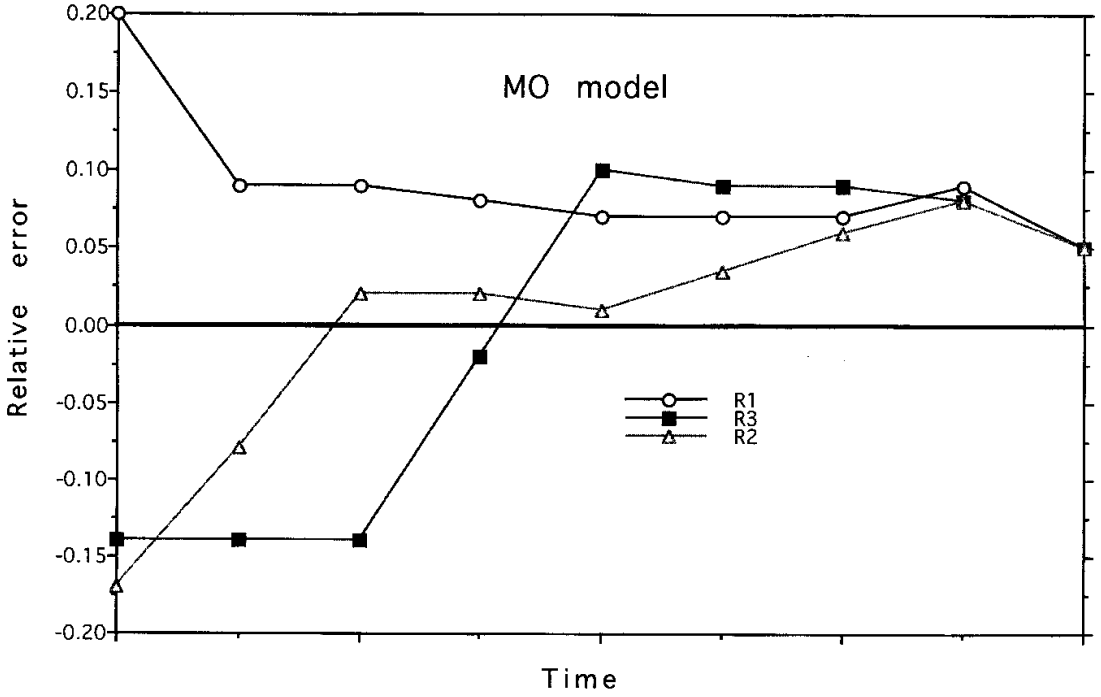

Figure 11.6 Relative error of long-term forecasts for three software releases (MO model).

complex for creating u-plots and PLR (prequential likelihood ratio) plots for more than just the next failure. Thus, they are primarily used when the prediction of the time to the *next* failure is important. As we mentioned earlier, the nature of failure event recording by the customer and vendor coupled with the data assimilation process usually leads to the retrieval of field failures in groups. Often the exact time of failure is not recorded—only the time when it was entered into a database. In any case, in considering the field data, we are not usually interested in when the very *next* failure event will occur, but rather how many will occur over time, and whether we can find subsequent interpretations for the parameters.

11.3.3.2 Computational diagnostics for model determination. Useful computational diagnostics typically fall into one of two categories: prediction error and overall fit. In previous discussions, we stated that accurate long-term predictions are an appropriate and important aspect of modeling field data. It is preferable to determine the accuracy of long-term predictions, or overall model behavior, especially when you wish to direct development processes based on the values of the model parameters. Therefore, certain short-term prediction diagnostics, such as prequential likelihood, are not examined in this section but are addressed in Sec. 4.3.

Computation of relative predictive error for reliability analysis was more formally stated in [Chan92]. Chan applied the idea of relative predictive error to both short-term and long-term predictions. Let ξ_i be

the relative error of the predictive model at time t_i. Then, for both long- and short-term predictions, we define the relative predictive error to be

$$\xi_i(\Delta) = \frac{\hat{\mu}_i(t_{i+\Delta}) - (i + \Delta)}{(i + \Delta)} \tag{11.1}$$

where $\hat{\mu}_i(t)$ is the forecast of the total number of failures by time t calculated at the ith failure time. For short-term predictions, typical values for Δ would be 5, 10, or 20, depending on what is meaningful to the organization and the reliability system. However, we must again remember that for field data, very short-term predictions are usually useless due to the nature of reporting and collecting data. Therefore, for long-term predictions, we are most often interested in the relative error of the predictions at some standardized time value t, where t is large so that comparisons can be made between releases. We may also be interested in relative errors for t_m where m is the total number of failures of the system (that is, $\Delta = m - i$).

In addition to relative predictive error, [Khos91] used the Akaike Information Criterion (AIC) [Akai74] as a model comparison method based on maximum likelihood. Although it is a means of indicating which model is better overall when compared relatively to another model, the values from the AIC metric are not as useful in determining how well suited any of the models are in absolute terms. That is, the AIC indicates closeness of the data to a relative distribution but does not, for example, convey how accurate any forecasts may be in the very comprehensible way that relative predictive error does.

Some traditional statistical modeling computational diagnostics should be avoided with software reliability models. One of these is the r-squared statistic, a value related to the lack of fit (sum of the squared errors). The r-squared statistic can be deceiving in this context since the r square is usually computed from a model based on a regression analysis using the failure intensity (which is, in turn, based on an approximation using grouped failures). In general, the larger the group size chosen (which is somewhat arbitrary), the larger the r square will be for the same data since the larger group size tends to smooth the data, thus eliminating variance in a trend. Comparisons of r-squared values from data set to data set are almost always suspect.

11.3.4 Data transformations

Transformations are a powerful mechanism for understanding data. Determining appropriate metrics and how the metrics are related is the key to understanding random phenomena.

We have seen that a linear relationship does not exist between the failure rate and the cumulative failures for DataSet 1. Can some trans-

form be used to examine other potential relationships? Table 11.3 illustrates several transforms that can be used to diagnose appropriate or inappropriate models. For example, the MO model exhibits a linear relationship between the log of the failure intensity and the cumulative failures, while the Duane model is linear in the log of the intensity versus the log of the cumulative failures. All the models in Table 11.3 use transforms of μ, λ, or τ. Also, all of the transforms can be easily calculated on most spreadsheets or statistical packages for quick analysis. Both SMERFS and CASRE support these types of data transformations. Some of these relationships, and the associated computational and graphical techniques required to perform this graphical diagnostic, are found in [Musa87, Jone91, Xie91a, Xie91b, Jone93]. The key relationship is the one between the failure intensity and the cumulative failures. Data transformations are especially effective when used in combination with graphs and plots.

We illustrate the transform approach using DataSet 1. The scatterplot with a regression line fit in Fig. 11.7 is the DataSet 1 failure intensity (λ) versus cumulative failures (μ) after taking a log transformation of the failure intensity. In this case, the log transformation creates an almost textbook linear fit. From Table 11.3 we see that the linearized MO model conforms with this transformation and may be the appropriate one to use. You should review Chap. 3 regarding the actual estimation of model parameters and derived variables. For a more complete discussion on curve fitting, regression analysis, and related diagnostics, see [Drap86].

11.4 Important Topics in Analysis of Field Data

In the case of a multirelease system, at different calendar times different software releases are installed at a different number of sites. This means that the usage intensity of a particular software load varies over calendar time and accumulates usage according to the amount of time the sites using the release have been in service. Therefore, from

TABLE 11.3 Linear Relationships in Reliability Growth Models

Model	Primary linear relationship	Other linear relationships (if any)	Parameters
GO model	$\lambda = \alpha\beta - \alpha\mu$	$\ln(\lambda) = \ln(\alpha\beta) - \alpha\tau$	$\alpha\ \beta$
MO model	$\ln(\lambda) = \ln(\lambda_0) - \theta\mu$		$\lambda_0\ \theta$
LV model	$1/\lambda = \beta_0 + \beta_1\mu$		$\beta_0\ \beta_1$
Duane (Crow) model	$\ln(\lambda) = f(\alpha\beta) - \frac{\alpha - 1}{\alpha}\ln(\mu)$	$\ln(\mu) = \ln(\beta) + \alpha\ln(\tau)$	$\alpha\ \beta$

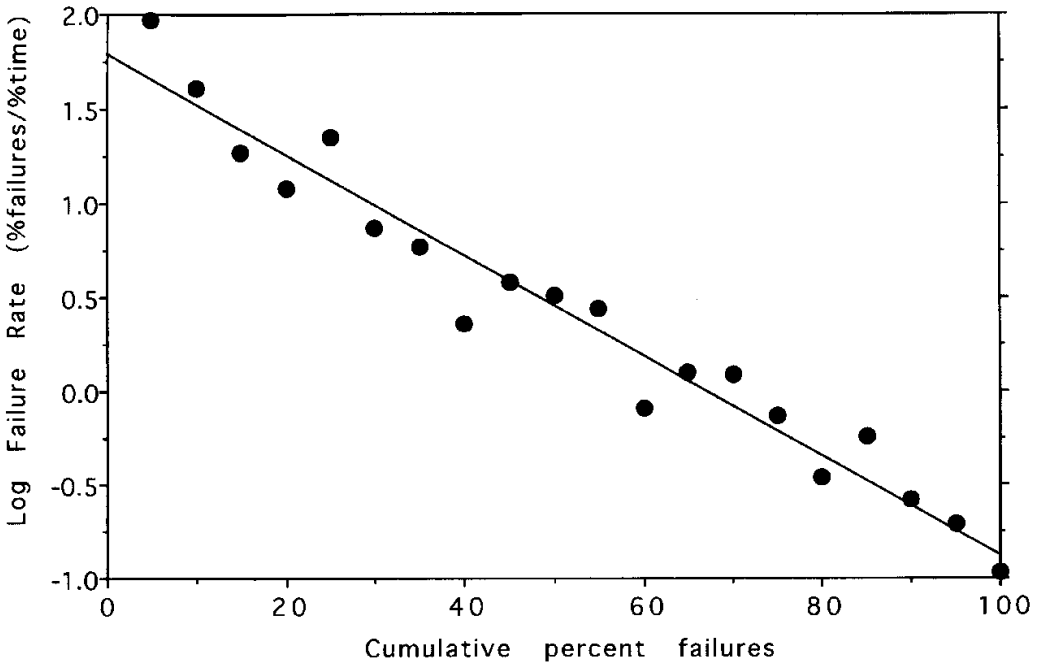

Figure 11.7 Log transform of DataSet 1 failure intensity.

both hardware and software viewpoints, in-service time is a more representative and relevant measure of usage than calendar time. However, in many cases, calendar time is a measure that better reflects the perception of users (such as the telecommunications companies) since calendar-time availability and degradation of services are very important from the customers' point of view. Simply stated, the context may dictate whether one or both viewpoints (calendar time and in-service time) are appropriate for analysis.

When discussing the quality of software in operational use, it is instructive to employ a classification based on the usage characteristics of the product and the nature and availability of the field data. As seen in previous chapters, it is well known that the software usage profile is a dominant factor that influences the reliability [Musa93], and that the software execution time is a better time domain than calendar time since it automatically incorporates the workload to which the software is subjected [Musa87]. The influence of measuring usage on reliability modeling as an alternative to calendar time is demonstrated by the example in this section. However, in practice, we may have to make a statement about the quality of the software without having direct information about its usage, and without having available a large number of failure events. Therefore, in the following sections, we will discuss three classes of field reliability data analysis: calendar-time, usage-time, and special-event analyses. In Sec. 11.8, we will discuss the related concept of availability.

11.4.1 Calendar time

Calendar-time analysis arises in situations where failures are reported only in the calendar-time domain and precise information about the usage of the software may not be available. We see this type of constraint in wide-distribution software—software that is developed for the purpose of selling on the open market to many customers, or for nonprofit distribution to anyone who wishes to install it. Its usage often builds to thousands, or even hundreds of thousands, of independent systems. However, direct monitoring of the usage rate of such software is not always feasible, or is not practiced. This is especially true of commercial wide-distribution software [Boeh89], shrink-wrapped or off-the-shelf software, and freeware. Examples of wide-distribution commercial software are Microsoft Windows, WordPerfect, DEC's Ultrix, commercial PC and workstation compilers, and freeware such as the GNU family of software products. We discuss calendar-time analysis in detail in Sec. 11.5.

11.4.2 Usage time

It should not be surprising that the many organizations that are prominent in the practice of software reliability engineering (SRE) deal with telecommunications and safety-critical applications.* Other application areas include reservation systems, banking transaction systems, database engines, operating systems, medical instruments, etc. For many of these systems, reliability is one of the most important, if not *the* most important, attribute of the system. This implies the need for accurate and detailed information about the system usage. Usage-time analysis can be performed when more precise information about software usage is available. This is often true for software that is developed for the purpose of selling to a specialized market such as the examples given above. Its usage may build up to hundreds or thousands of independent systems, yet the users of the software are known and well documented, and direct monitoring of the usage rate of the software is feasible and is practiced. We discuss usage-time analysis in detail in Sec. 11.6.

---

* For example, *telecommunication systems, networks:* ALCATEL, Bell Laboratories/AT&T, Bellcore, BNR/Nortel, Ericsson/GE, Fujitsu, IBM, Motorola TELEBRAS; *advanced avionics and space systems:* Boeing, CNES, Hughes Aircraft, LORAL (ex IBM Federal Systems), JPL, NASA, USAF; *other systems:* Cray, Digital, Hewlett-Packard, Hitachi, Intel, Sun Microsystems, StorageTek.

11.4.3 An example

The following example helps underscore the issues driving the above classification. Figure 11.8 shows the actual field data for a large-scale limited-distribution telecommunications product. We plot the concurrent changes in the number of installed systems of a particular software release (DataSet 1 on the Data Disk) over calendar time, the corresponding failure rate in terms of calendar time (failures per week), and failure rate per system in-service week. Note the dramatic difference between the failures per calendar week and the failures per system in-service week.

From Fig. 11.8, we see that the calendar-time failure rate is initially low (indicating apparent high reliability), then begins to climb (apparent reliability degradation), and finally reaches a peak just before the deployment reaches its peak. A naive analyst might mistakenly conclude that a disaster is in the making. In fact, the system is behaving as it should—the problem is an inherent deficiency in the failure rate metric. As the rate of deployment peaks, the reliability appears to improve dramatically, and the failure rate drops steadily thereafter.

However, we see a different picture when we examine the failure rate in terms of failures per system in-service week (that is, normalized with respect to the deployment function) or per usage load on the system. The normalized failure rate is initially high, but then decreases dramatically in the first few weeks after the system has been deployed. As the deployment curve peaks, the reliability growth may slow but reliability continues to improve.

Obviously, the model that describes the failure behavior of this system will depend very strongly on whether we have the actual system

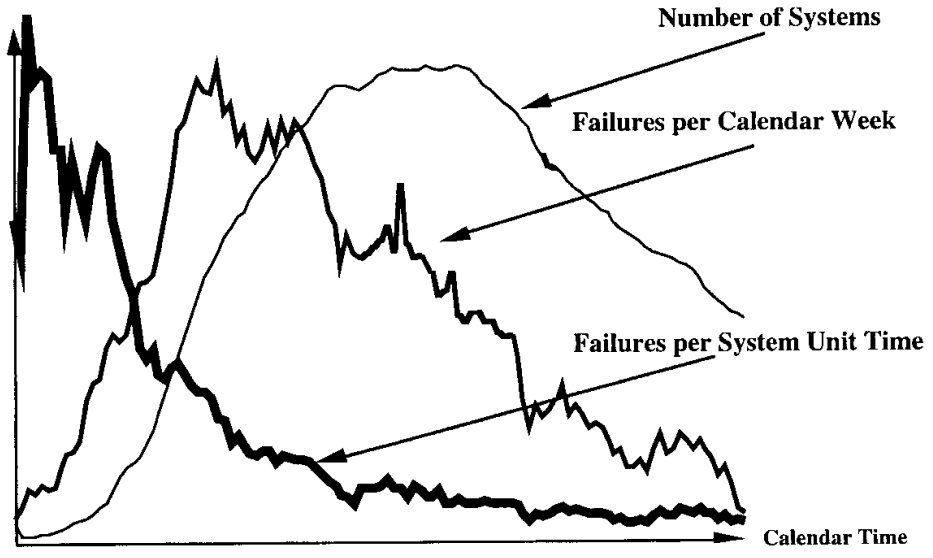

Figure 11.8 Influence of usage on computed failure intensity.

usage information or not. The number of failures per calendar week is a direct function of the true reliability of the system *and* the deployment function of the system. Reliability growth may be difficult, if not impossible, to discern from the calendar-based view. While failures per calendar week may be a natural, and important, metric to a customer service organization, it is usually far from suitable for making inferences about the reliability of the system [Musa87].

11.5 Calendar-Time Reliability Analysis

The principal characteristic of wide-distribution software is that it is used by many users at many customer sites. This software lives in the world of multiple releases, and for this type of software, by definition, we often do not know who the users are or how they use it. Although we may know how many licenses there are, we may not know how intensively each copy is used. When dealing with wide-distribution commercial software we often have a large user base with each user experiencing his or her own level of reliability. Some users may run for months without a disruption, while others may run for only a few hours before running into a problem. It all depends on the user's software usage profile. Yet, despite the fact that reliability of a software system is important to customers of commercial software products, they generally do not keep good records on execution time, and they seldom report on their reliability experience. What they do report to software development organizations is the occurrence of specific failures, with the expectation of getting the underlying defect(s) fixed so that the failures do not reoccur. This is possibly why many commercial software development organizations focus on the number of remaining defects rather than reliability, or mean time to failure, as the measure of software quality.

[Musa87] discusses the advantages and disadvantages of the calendar-based analysis in great detail, and shows that although a general Weibull-type failure intensity model can describe calendar-time system behavior, the fit is often inferior to the one obtained using execution-time-based intensity. But it is also pointed out that in practice managers and users may be more attached to the calendar-time domain since it is closer to the world in which they make decisions.

The following case studies illustrate practical analyses of calendar-time data. We consider unimodal models, and the use of calendar-time-based failure intensity to focus on the user perception of the software quality. In the first case study, analysis is biased toward the view software developers may take when dealing with field defects, while the second study is strongly biased toward the user view. We leave it as an exercise for you to research and discuss the bias, if any, in the remaining examples.

11.5.1 Case study (IBM Corporation)

One approach to analyzing field data for wide-distribution commercial software is described in detail in [Kenn92, Kenn93a, Kenn93b]. The idea is to decouple the concept of individual system reliability from the notion of how many defects are left in the code as a whole, and to use the latter to quantify the quality of the software through a Weibull-type field-defect model. The approach is based on the assumption that for the software developers the failure count is of special interest because it implies a certain software maintenance workload, and that by estimating the number of defects remaining in the code it is possible to distill the experience of thousands of users down to one number that characterizes the quality of the software product as a whole.

[Kenn93a] developed a calendar-time model for a multirelease product using Trachtenberg's general theory of software reliability [Trac90]. Since direct usage information was not available, it was assumed that, within the period most interesting for the study, the workload on the system as a whole increased as a power function of calendar time. The general form of the failure intensity for the resulting Weibull field-defect model is

$$\lambda = N\left(\frac{\alpha}{\beta}\right)\left(\frac{t}{\beta}\right)^{\alpha-1}\exp\left(-\frac{t}{\beta}\right)^{\alpha} \tag{11.2}$$

where N is the initial number of defects in the software, t is the calendar time and α and β are the Weibull parameters. Under the assumptions made in the study, λ is both the average defect discovery rate and the average defect removal rate. Parameter estimation for α and β was done by maximizing their conditional likelihood. The estimate of N was obtained through substitution of the estimated parameters α and β and of the number of observed failures at the point of censoring in to the cumulative distribution function. Of course, other approaches to estimation can be used.

In [Kenn93a] two data sets from releases of established IBM software products were used to demonstrate the model. One data set contains 36 months of defect-discovery times for a release of *controller software* consisting of about 500,000 lines of code installed on over 100,000 controllers. The other contains 24 months of defect-discovery times for a release of a commercial *software product* consisting of about 1 million lines of code installed on 10,000 systems. The defects are those that were present in the code of the particular release of the software and were discovered as a result of failures reported by users of that release, or possibly of the follow-on release of the product.

The first of the above data sets is on the Data Disk as DataSet 2. We illustrate the results in Fig. 11.9. The figure shows the number of

unique defects reported per month against the month after the first customer shipment and the associated Weibull model, which has an S-shaped cumulative distribution function. The downside of using an S-shaped model, such as the Weibull field-defect model, is that, in this case, a reasonable and stable estimate of N could not be obtained before about 16 months of data were available for the *controller software,* and about 18 months of data were available for the *software product* set, although preliminary (and less accurate) estimates could be obtained as early as 10 to 12 months after release of the software to the customers.

It is a common experience that when a new software release is made available for customer use, the reporting of field problems initially rises to a peak and then tails off as the release ages [Uemu90, Ohte90, Yama91, Mart90, Leve90]. When the next release is shipped to customers, we again see the field problem reports rise. It turns out that this increase is not totally due to defects inserted into the new release. In addition to discovering the defects added through the new release, the new release often stimulates discovery of the latent defects, that is, the defects that were inserted into the prior code releases but never discovered. This phenomenon aggravates the stability of a new release in its early days, and makes predictive modeling difficult. We call the phenomenon the *next-release effect* [Kenn92, Kenn93a, Kenn93b].

The next-release effect can be seen in Fig. 11.10. The figure shows an overlay plot of the frequency of reported problems for several releases of the *controller software.* The data are smoothed using a three-point symmetric moving average. All releases use common timescale, months after

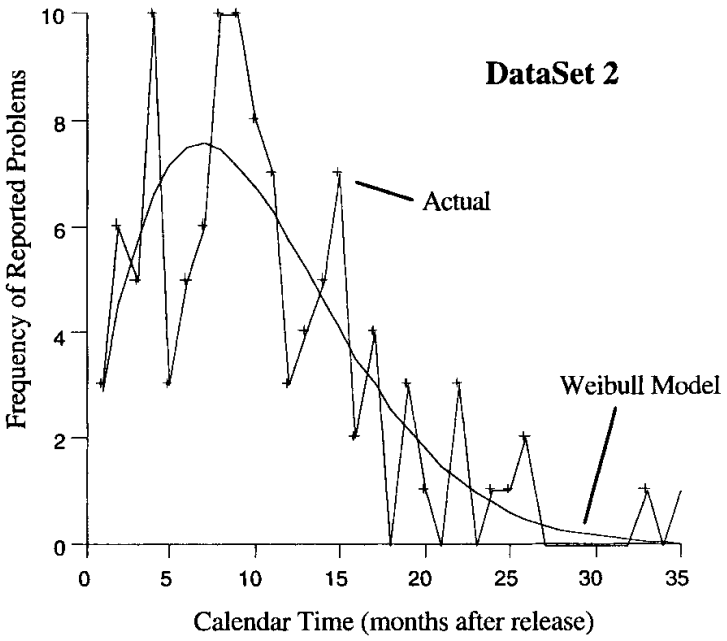

Figure 11.9 Frequency distribution and its Weibull model (DataSet 2).

release X was shipped. The time span covers six releases, but for simplicity we show only three of them: $X, X + 1$, and $X + 2$. We see the effect subsequent releases have on release X as multiple peaks in the release-X-associated problems around months 9, 12 to 13, and to a lesser extent at months 16 to 19, 25, and 34. These peaks were correlated with the switchover of the customers to the next release, and the onset of reports from the next release of the product [Kenn92]. The release of new code stimulates discovery of defects in the prior releases of the code. For example, the defects present in the code developed for release X were reported as problems during field use of release $X + 1$ and traced back to release X code. The next-release effect is further discussed in the context of the space shuttle flight software failures (see Sec. 11.7.2).

It is interesting to note that the above analysis, from the data collection perspective, requires that the detected defects be counted against the release into which they were inserted, which is not necessarily the release on which the first failure occurred. This means that a mapping would have to be established between the field failures and the version from which the corresponding faults originate, information that may not be readily available in many organizations.

11.5.2 Case study (Hitachi Software Engineering Company)

The Japanese software industry has a long and distinguished record of considering the quality of software they ship to their customers as a

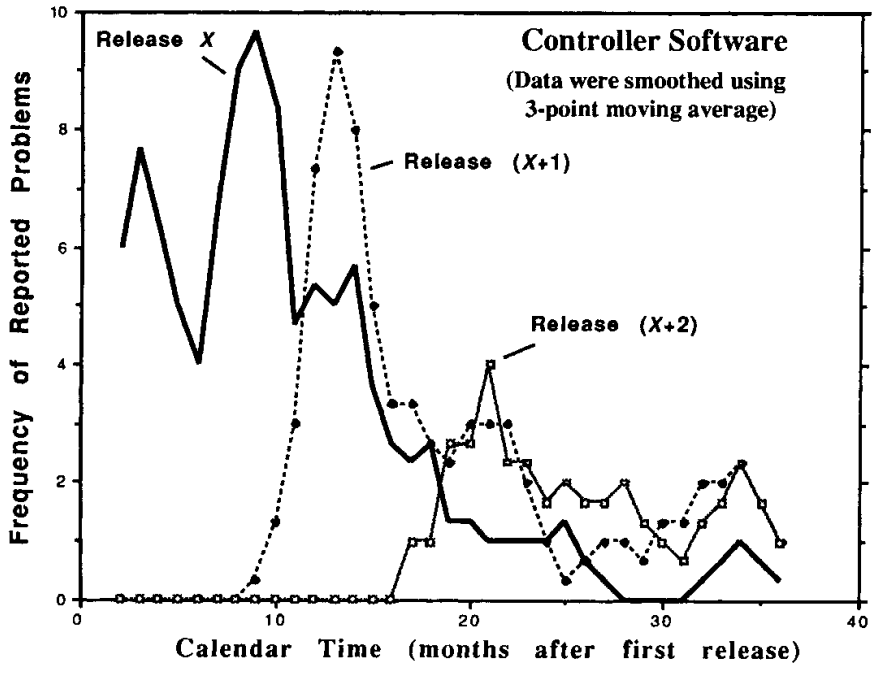

Figure 11.10 Smoothed frequency distribution for releases X, $X + 1$, and $X + 2$.

very serious issue [Naka92, Ishi91]. Good examples of such companies are Fujitsu and Hitachi.

For instance, Hitachi Software Engineering Company Limited, a part of Hitachi Limited, is one of the largest software houses in Japan. About 20 percent of the systems developed at Hitachi Software have 5 million lines of code, while the average size of a system is about 200,000 lines of code. Hitachi Software uses statistical control to improve both the quality of the software they ship and the quality of the process they use to develop it. During the development, Hitachi Software engineers predict the defect content of their product and derive the test-stopping criteria using a Gompertz curve fault model. After the product is released, software field data is routinely collected, and the information is fed back into the software development process to effect further improvements [Onom93, Onom95]. The overall goal is customer satisfaction and the key value is the number of failures that a customer may experience per calendar month.

Hitachi Software tracks the number of system failures detected at customer sites. This information is transformed into the *field failure ratio,* the number of system failures per month per thousand systems [Onom93]. Figure 11.11 shows the percentage change in that ratio over a 13-year period (DataSet 5 on the Data Disk). The ultimate target is zero field defects, and more immediate targets are set with the aim of having as small a number of problems as possible emerge at customer

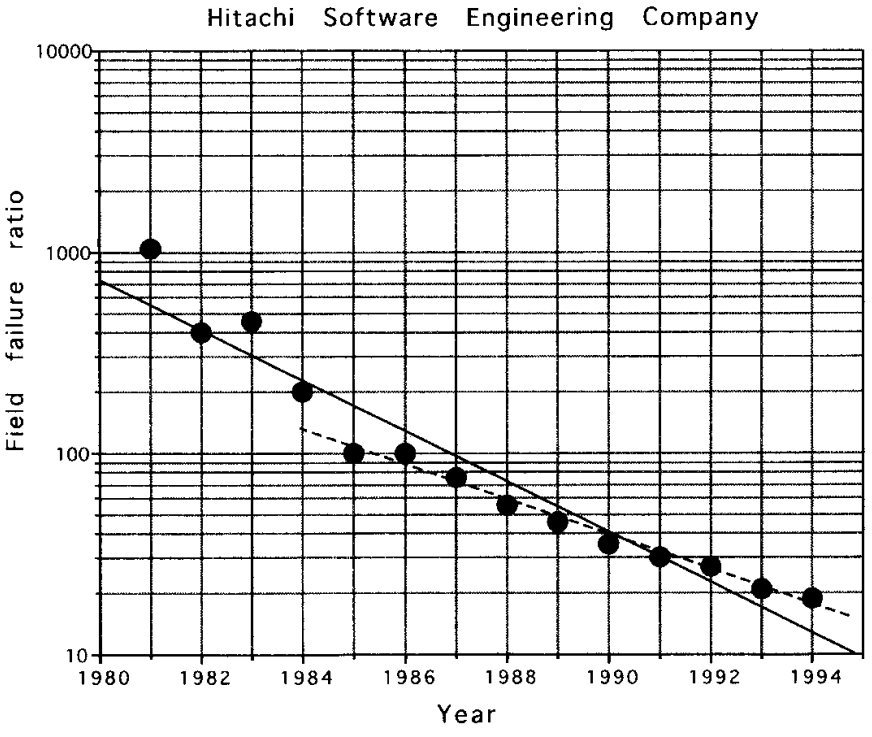

Figure 11.11 Field failure ratio.

sites. To emphasize this, Hitachi Software measures the productivity of its engineers not in terms of the number of lines of code produced per hour, but in terms of the number of debugged lines of code per hour. For example, in 1990 only about 0.02 percent of the faults detected during a project life cycle ended up as field failures [Onom93, Onom95].

11.5.3 Further examples

AT&T. An interesting calendar-time birth–death model was developed by Levendel [Leve87, Leve89, Leve90, Leve93]. Levendel defines and models a set of laws that govern the software defect detection and removal process. He uses these models to describe and predict the behavior of software field failures. The model includes analyses for incremental defect detection, removal, and reintroduction rates. The underlying assumption is that the defect introduction and removal is governed by Poisson distributions. For example, his simulation shows that for every three defects repaired as much as one defect may be reintroduced. The defect rate (intensity) model envelope is a *unimodal Poisson-type shape.* Levendel also discusses quality estimators such as the number of defects to repair, current number of defects, testing coverage and repair intensity, and testing process quality and effectiveness. The laws and models were applied to a large telecommunications switching product (5ESS), and were found to be good descriptors and reasonable predictors of the system behavior.

TELEBRAS and ALCATEL. Several interesting approaches to analysis of failure data and calendar-time usage information can be found in the extensive studies of field reliability and availability of the TELEBRAS TROPICO R digital switching systems (e.g., Chap. 10, [Mart90, Mart91, Kano91a, Lapr91]) and of the ALCATEL E-10 system ([Kano87]). For example, [Mart91] applied trend tests, and fitted the exponential and gamma [Yama85] models to the TROPICO-R 4096 system test and operational data (on the Data Disk, DataSet 4). [Mart91] experimented with the removal of some initial data to avoid spurious results due to transient behavior usually found in the initial recorded usage period. You are encouraged to calculate and plot DataSet 4 intensity, the corresponding Laplace trend graph, and then to attempt to fit MO, GO, and hyperexponential models and discuss the results (first to all data, and then to the data after time unit 10). Kanoun et al. reported on a similar study performed on the TROPICO-R 1500 system (DataSet 3) [Kano91].

The validity of the hyperexponential reliability model was explored using the data from the ALCATEL E-10 switch [Kano87] and from the TROPICO-R system [Lapr91]. [Lapr91] also illustrated how trend

tests help to partition the observed failure data according to the assumptions of the reliability growth models. It is worth noting that TROPICO-R data (DataSet 3) appear to show the influence of the usage ramping, but a direct hyperexponential fit treats that as an initial instability and essentially overrides it. On the other hand, the approach taken by [Mart91], where the initial data were discarded, simply removes the transient behavior. We want to stress that, although in some situations this is a valid approach,* in general, any elimination of early data (or other outliers) has to be done with great caution and with considerable expert help because it can lead to models that have no predictive validity. Some related results are given in Chap. 10.

11.6 Usage-Based Reliability Analysis

As vendors and customers form closer alliances due to stringent reliability expectations, usage data will be more accessible. Currently, Bellcore [Bell90b] requires that telecommunication vendors in the United States systematically record software usage and a variety of metrics that quantify the quality of software releases. In fact, it is likely that in the near future, industries such as medical software [Frie91] will be subject to similar requirements from some outside agency. In this section, we show how software reliability analysis can be conducted when sufficient usage information is available.

11.6.1 Case study (Nortel Telecommunication Systems)

The reliability behavior of Nortel's DMS (Digital Multiplexing System) software was initially studied by Jones [Jone91, Jone92]. The focus was the comparison of the predictive accuracy and goodness-of-fit of various models along with a comparison of parameter estimation methods. The ultimate goal was to determine appropriate modeling methods for Nortel's DMS software. A comparison was made of the ability of various models to forecast the total number of field failures and corresponding faults. In addition, the reliability model aptness or goodness-of-fit during alpha and beta test was examined. Finally, given a model, a comparison was made of the two popular methods for estimating parameters (maximum likelihood and least squares).

* In situations where the failure observations cover a long time period of time, old data may not be as representative of the current and future failure process as recent data. In such cases, justified data aging may be an appropriate approach (see Secs. 11.7.3 and 11.8).

By examining several releases over a period of time, [Jone92] used a variety of numerical and graphical diagnostics to conclude that the MO model was the most appropriate model for the DMS software releases both in the field and during alpha and beta field testing. An example using release R8 (a large software release of a telecommunications product) is shown in Fig. 11.12. In Fig. 11.12*a*, we see that the failure intensity does not appear to be linear with respect to failures. However, the log transformation of the failure intensity shown in Fig. 11.12*b* does appear to greatly linearize the data, as well as homogenize the variance of the failure intensity. From this and other analyses, it was concluded that the MO model was most appropriate for this software system.

In a follow-up study, parameters of the field MO model were related to metrics of the development cycle and to the deployment characteristics of the software [Hude92]. It was found that for DMS systems, the decay parameter θ of the MO model was directly related to the number of systems deployed. If a release had wide deployment, the decay rate of the failure intensity was less than when a release had limited deployment. Also, the initial failure intensity (parameter λ_0 of the MO model) in the field was found to be directly related to the failure intensity measured during alpha and beta test.

11.6.2 Further examples

AT&T telecommunication systems. [Ehrl90, Ehrl93] examined system test and field data from the beta test of the AT&T System T software.

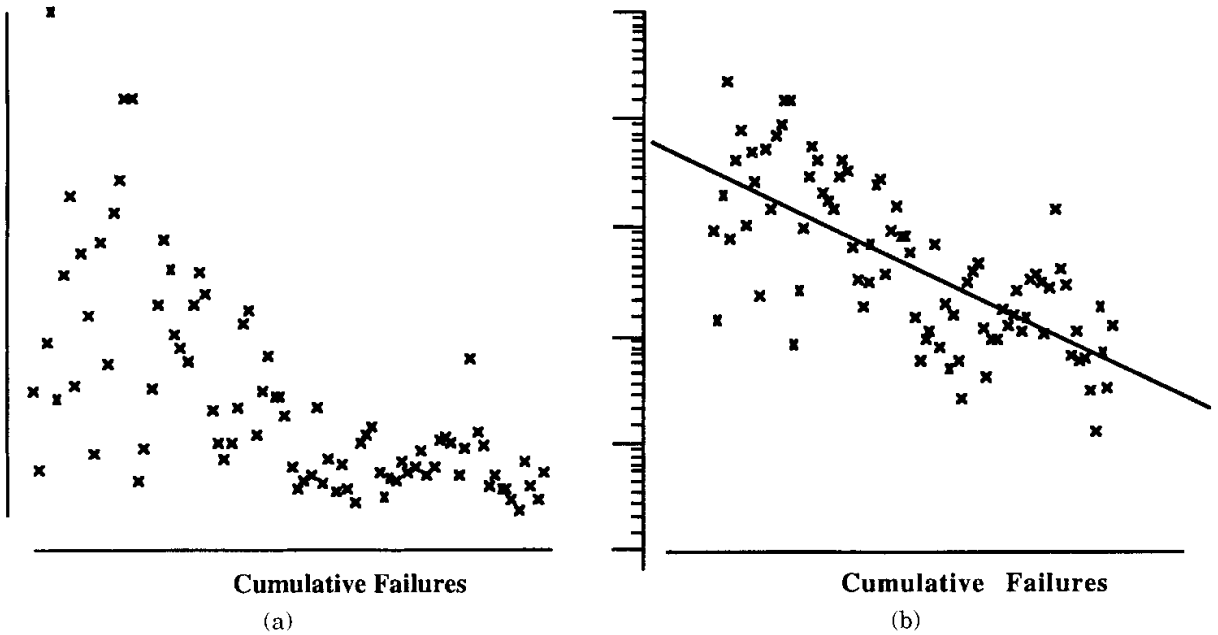

Figure 11.12 (*a*) Reliability modeling of the Nortel DMS software. Graph of failure intensity for R8 during alpha- and beta-test phase. (*b*) Reliability modeling of the Nortel DMS software. Graph of log failure intensity for R8 during alpha- and beta-test phase.

The more recent work [Ehrl93] is particularly interesting since the authors demonstrated a procedure for testing Poisson process assumptions. Poisson processes are often used when modeling reliability growth. Their candidates for modeling their failure data were the GO, MO, and Duane models. The GO model was chosen for System T since it had the smallest fitted mean square error.

VAX/VMS operating system. [Tang92] examined the software reliability of the VAX/VMS operating system on two VAX clusters. Analysis considerations included examining distributions of the time to failure, time to repair, and time between failures. Also of interest were software failures that were hardware-related and estimates of unavailability due to software. For the systems under study, the unavailability due to software was on the order of 10^{-4}. As a contrast, telecommunications systems are usually required to have an unavailability due to all causes of less than 10^{-5}. The authors of [Tang92] used a modified GO model in which they assumed that an asymptotic steady-state constant term existed for the failure intensity. It is interesting to note that this model is similar to the hyperexponential model [Kano91].

CNES space and avionics system. [Vale92] examined two data sets related to space systems that had operational or field failure data. The goal for this study was to determine which reliability growth model would be the most appropriate for the authors' software systems. They initially selected five models: Musa model (basic execution time), Littlewood-Verrall, GO model, Weibull (Duane) model, and the MO model. They were also interested in determining if there were relationships between the development environment or development metrics and the appropriate software reliability growth model. The diagnostics used in the model determination included Kolmogorov-Smirnov test on a u-plot and evolving u-plot along with the relative predictive error. Their data suggested that field failures from one project in one category were best modeled by the MO model, while field failures from another project in a different category were best modeled by the Duane model.

Bellcore software quality measurements. A very interesting issue is how reliability models relate to the overall software process [Naka92, Paul93] and how classical quality control techniques, such as quality trend charts [Hoad81, Hoad86], can be applied to evaluate the quality of a series of software product releases. [Weer94] examined two sets of data belonging to two telecommunications products in the light of the Hoadley's Quality of Measurement Plan (QMP). Both sets contained data on three releases of the product. In their paper the authors describe how failure data from software processes can be displayed

using QMP trend charts, and how the approach can be used to compare a number of software releases to see if there is an increase or decrease in the software quality over the releases, where the quality is measured through the number of unique field failures (faults) and the fault density (faults per 10,000 lines of code). To make fault estimates, the authors experiment with exponential, Pareto, and Weibull distributions for fault detection times (the time scale is "system months of field operation"), and they use a Bayesian methodology to combine the data from all releases to produce quality index estimates. This type of analysis will gain in importance as software process maturity of organizations increases, and as the organizations start using their test and field deployment failure data to control software field maintenance processes and provide immediate and active feedback to the process of developing the next release of their software product [Vouk93b].

11.7 Special Events

Some classes of failures may be of special interest, and may be considered more important than others. These could be failures that are categorized as having life-threatening or extremely damaging consequences, or failures that can be very embarrassing or costly. The need to recognize early the potential of a software-based system for special-event problems is obvious. How to achieve this is less clear. In general, it is necessary to link the symptoms observed during, say, software testing phases or early deployment phases with the effects observed in the later operational phases.

In that context, the key is identification of these failure modes, and of the events that lead to these failures. Failures modes that are absolutely unacceptable should *not* be analyzed using only probabilistic methods since these methods are inherently incapable of assuring the level of reliability that is required for such systems. Some other techniques, such as formal methods, should be used to complement the analyses [Schn92b]. Ultrahigh reliability systems pose special problems and require dependability assessment techniques. A discussion of these issues can be found in [Butl93] and [Litt93].

However, special-event failures to which one is willing to attach a probability of occurrence (say, above 10^{-7}) may be analyzable through the concept of risk. This concept forms a bridge between the probabilistic reliability aspects and the critical and economic considerations of any system [Ehre85, Fran88, Boeh89]. A risk model identifies a set of software reliability engineering indicators or symptoms, and relates them to the expected behavior of the software in the field. Once risks have been identified, analyzed, and prioritized (risk assessment), a risk control strategy has to be defined and implemented. In practice, risk assessment

and control requires a very thorough understanding of the problem area, solution alternatives, and corresponding impacts. Furthermore, the process is invariably complicated by the fact that the probability and/or loss estimates are subjective, at least to some extent, and that our information about the system states and associated impact likelihoods is never perfect. For example, we have to take into account the probability that our risk control decisions, based on the computed risks, will fail to avert the risk, or will be wasted since the risk would not have materialized in any case. Before applying software risk assessment and control techniques, you are urged to consult the excellent works of [Boeh91, Boeh89, Char89, Fran88, Ehre85] on software risk analysis, the papers and references in [IEEE94] on software safety, and the *Human Reliability and Safety Analysis Data Handbook* [Gert94].

An example of a special event that could be regarded through the probabilistic prism is an FCC-reportable failure. In part owing to a series of operational problems that have embarrassed the switching industry in the past several years,* FCC has issued a notification to common carriers regarding service disruptions that exceed 30 minutes and affect more than 50,000 lines. Since March 1992, any outage of this type needs to be reported to FCC within 30 minutes of its occurrence [FCC92]. These FCC-reportable events are relatively rare, but such outages may have serious safety implications.† Since the information itself can command considerable public visibility and attention, such failures may have serious business implications as well. An example is an advertisement that appeared in the *USA Today* on April 15, 1993 [USAT93]. This AT&T advertisement woos 800-service customers by comparing the AT&T and MCI reliability performance over the previous year in terms of, among other things, lost and abandoned calls and the number of FCC-reported outages.

11.7.1 Rare-event models

The key issue is the probability of occurrence of rare events. Computation of the probability of rare software events is not a solved problem, and perhaps not even a fully solvable problem. However, whatever results are available must be presented not as a point estimate but as a range or interval: for example [lowerbound, upperbound]. Often, 95 percent confidence interval is used. We present some very simple models which serve to highlight the issues involved, and indicate the difficulty of the problem.

---

* For example, the January 1990 AT&T outage [Lee92a] and the DSC signal transfer point fiasco [Wats91].

† For example, through impact on emergency services such as 911 calls.

11.7.1.1 No-failure model. It can be shown (e.g., [Dura84, Ehre85, Howd87]) that if N representative (operational profile) random test cases are executed and no failures are found, then an upper bound, p_u, on the failure probability of the system, at α confidence level, is given by the following expression:

$$p_u = 1 - (1 - \alpha)^{1/n+1} \tag{11.3}$$

This is the distribution function of the *modified* geometric probability mass function which is used when one counts the number of trials before (but *not* including) the first "failure" [Triv82]. Typically, $\alpha = 0.95$ (95 percent confidence bound). For example, given that we have run for 10,000 in-service hours without experiencing a single FCC reportable failure, an upper 95 percent confidence bound on the probability of failure per in-service hour is

$$p_u = 1 - (1 - 0.95)^{1/10001} = 1 - 0.9997 = 0.0003 \tag{11.4}$$

So the model, given continuation of execution in the same environment, is [0.0,0.0003] failures per in-service hour (to one significant digit). A more sophisticated analysis, based on Bayesian estimation, is offered by [Mill92]. Equation 11.4 is then a special case where prior knowledge about the quality of the system is not incorporated into the calculation.

11.7.1.2 Constant-failure-rate model. If some failure information is available, and it can be assumed that the failure rate, or failure intensity, is constant,* then we deal with two cases: (1) a gamma (exponential) distribution, if the number of failures is fixed but the total exposure time is a random variable; or (2) the Poisson distribution, if the number of failures is a random variable but the total exposure time is fixed. In both cases, standard statistical confidence bounds for these distributions can be used to evaluate the information.

The simplest model is the one where we estimate the probability of the undesirable events based on the counts of these events:

$$P(S_f) \simeq \frac{n_f}{n} \tag{11.5}$$

where n_f is the number of failure events, and n is the usage exposure expressed as the number of intervals in which we wish to estimate.

For example, given that a telecommunications organization experiences three FCC-reportable failures in one year [USAT93], and if we

* Or, at least, a relatively slowly varying function of time.

assume that the failures are mutually independent and the rate is relatively constant,* then (using 95 percent Poisson bounds [Triv82]) in the following year we may expect between one and seven FCC-reportable failures. Similarly, if the constant annual failure rate is 13, then the 95 percent confidence-bound model is [7, 19] FCC-reportable failures per year.

11.7.1.3 Reliability growth. If the usage rate of the product is growing, but its quality remains approximately the same or grows at a lower rate than the product usage, then although per-site or per-system failure rate may be roughly constant (or may even be improving), the overall number of reported problems will grow. In that case, it is necessary to model the per-site failure rate. For example, let function $S(t)$ describe the number of sites that use a particular release of a product at some calendar time t (see Fig. 11.2). This shape can often be described using a Poisson [Leve90], or perhaps Weibull-type, envelope, such as

$$S(t) = K \frac{\alpha}{\beta} t^{\alpha - 1} e^{-(t/\beta)^{\alpha}} \tag{11.6}$$

Combined with historical information about the usual position of the envelope mode, and other model parameters, and the marketing information about a release (e.g., the total number of sites expected to run this release), it may be possible to predict $S(t)$ relatively early in the life cycle of a release. This yields an estimate of the overall load on the software release. If this function is then combined with the one describing the quality of the release, it may be possible to make early and accurate predictions of the outage rates.

An example of the quality function, $q(t)$, may be the annual variation of per-office failure rate, perhaps along the lines seen in Fig. 11.11 for Hitachi Software. Note that the problem is somewhat different from traditional time-dependent reliability modeling. The assumptions are that the exposure time is the calendar time, that $q(t)$ is a slowly varying function of time compared to $S(t)$, and that $S(t)$ is relatively independent of $q(t)$. In many practical situations this may be true, and $q(t)$ of a release may be fairly constant over the life cycle of a release (except, perhaps, in the very early field release stages [Cram92]). Hence, although the solution presented by Eq. (11.7) is exceedingly simple, it may be adequate; that is

$$f(t) = S(t) * q(t) \tag{11.7}$$

* Or that it is at least a function that changes very slowly over a two- to three-year period.

may estimate the average number of field outages per unit time.* Whether $f(t)$ estimates the number of outages for a release or for all active sites will depend on whether $S(t)$ represents a single release or is a combined function. The confidence bounds would have to be computed based on the standard error of the site estimates and the $q(t)$ error.

Different models can be used to described the growth of quality. It may also be possible to use all available failure data, not just the special-event data, to estimate reliability growth, and then make assumptions about the proportional growth in the reliability with respect to the special events. A more detailed discussion of this approach can be found in [Schn92b].

11.7.2 Case study (space shuttle flight software)

Space shuttle onboard flight (SSOF) software is an example where a team of experienced software reliability engineers evaluated a number of reliability models and selected the models that best matched the rare-event failure history of that software [Schn92b, Schn93c].

The team carefully evaluated all assumptions on which the models were based and made sure that all restrictions they imposed were accounted for in the analysis of the real software. The execution time of the software under investigation was estimated using test records of digital flight simulators and records of the actual shuttle flights. The failure data collected over a 12-year period were used to validate the models. All detected faults were considered in any operational-like execution, whether the user was aware of the fault or not [Schn92b]. The events were quite rare—at most several per usage-year of a module, and decreasing in frequency. The researchers fitted different models, made predictions, and compared the results to the actual data. They defined and applied different data aging criteria† to account for the fact that old data may not be as representative of the current and future failure process as recent data. In fact, they found that if they gave lower weight to, or even excluded, earlier data they could get more accurate predictions of future failures [Schn93c]. The Schneidewind model provided the most accurate fit over the investigated period.

The data used in the preceding modeling is on the Data Disk (DataSet 6). The data was provided by NASA [Prue95] and reflects the flight software failure history from January 1, 1986, through December

* The rarer the event, the longer the time period.

† Examples of data aging include moving average and exponential smoothing. See Sec. 11.8 for application of data aging in availability studies.

1, 1994. The data sets include information about the severity of the failures, information about the circumstances in which a failure was observed (TESTING, TRAINING, FLIGHT), and the information about the software version in which the failure was observed and in which it was originally introduced. Figure 11.13 shows the cumulative failure count over that period, and Fig. 11.14 shows the total yearly failure rate by severity. The exposure time is calendar time. These data show that only one failure was recorded on a flight and it was of low-severity level 3. The remaining 99 failures were found during planned verification testing and training use of the software. The NASA standard, and experience for the past five years, has been that SSOF software fault rates are in the range 0.1 to 0.2 faults of any severity per 1000 lines of changed code [Prue95]. Furthermore, all but one failure were detected prior to flight, and no potential severity 1 or 2 failures have *ever* occurred in flight. The severity 3 flight failure involved a benign annunciation issue analyzed postflight. This success is less surprising when one realizes that the software organization that develops SSOF is one of the very few software organizations in the world that has been classified as a level 5 on the SEI software process scale [Paul93, Kits91].

Severity classification used for SSOF failures is rather unique to avoid confusion with many other severity definitions used on other software projects [Prue95]. Full definitions are given on the Data Disk in the DataSet 6 file. When discussing the loss of or injury to the crew or vehicle, even a single severity 1 problem may be considered as a seri-

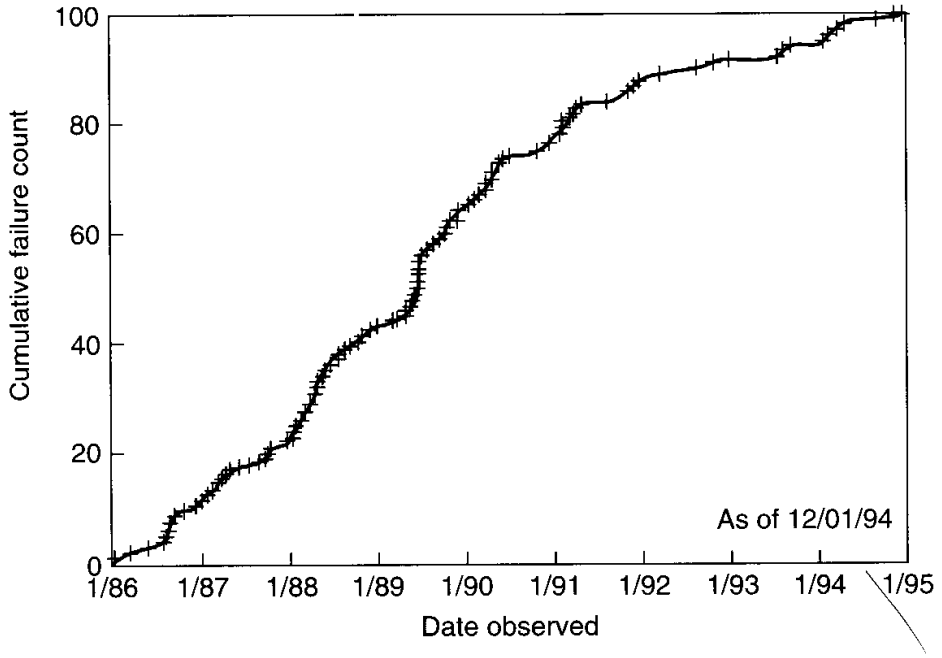

Figure 11.13 Cumulative failure count for SSOF software over a nine-year period [Prue95].

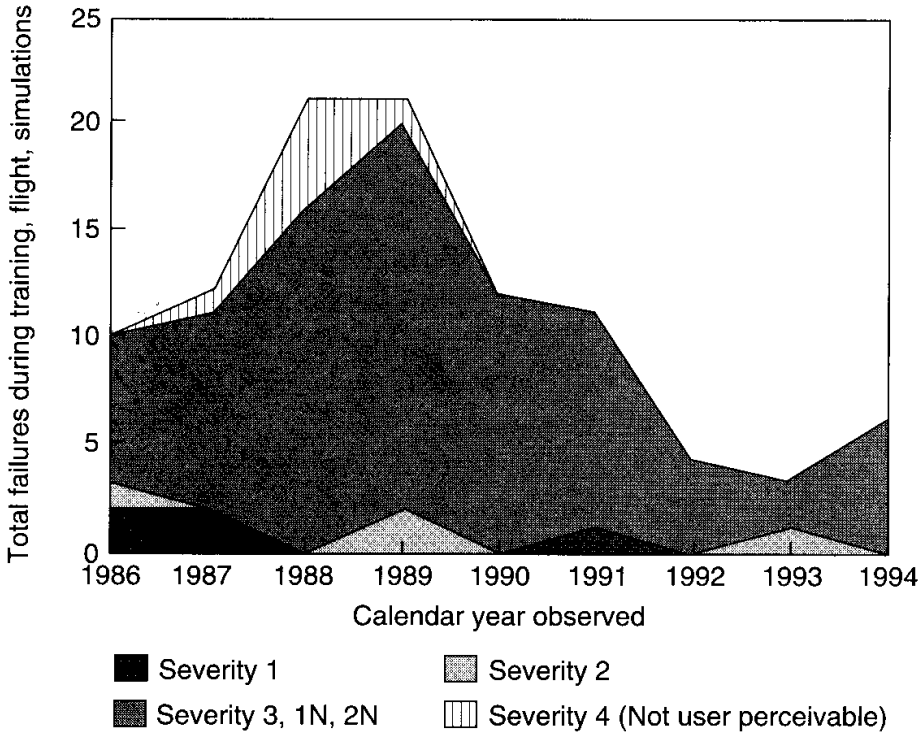

Figure 11.14 SSOF software yearly failure rate over the past nine years, and the severity of the observed failures [Prue95]. Since 1985 only one failure has been experienced in flight due to a coding error. That failure (in 1989) involved a benign annunciation issue analyzed postflight (severity 3).

ous oversight and cause for concern. Hence, SSOF severity definitions are very conservative. NASA defines as severity 1 any problem that has even the remotest probability of ever occurring in normal operational use of the shuttle. The experience is that severity 1 and 2 problems almost always require highly off-nominal shuttle operations or multiple hardware failures within millisecond windows to result in an actual failure. Although these situations have very low probability of occurring, NASA takes severity 1 and 2 failures very seriously and requires detailed analysis and corrective action on each.*

Figure 11.15 illustrates the next-release effect as observed in the SSOF software. The filled circles represent the failures, attributed to the first system release (REL A) of the code, that were precipitated through the execution of a later release. The release execution parallelism derives from concurrent testing and use of several releases under different conditions.

It is important to note that the Schneidewind model was not used as the principal indicator of software reliability, but only to add confi-

* This is also done for shuttle hardware failures.

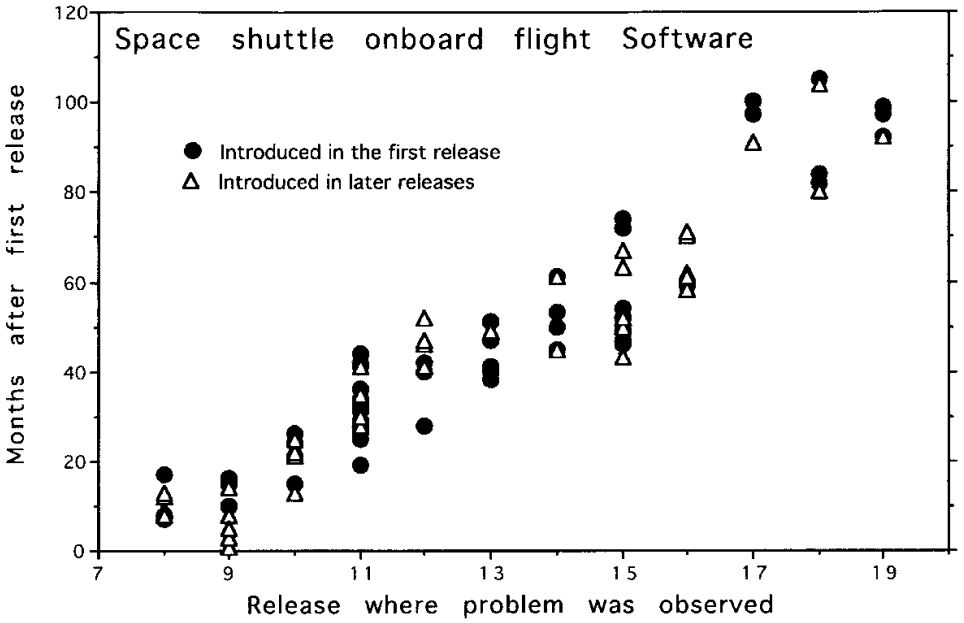

Figure 11.15 This dot plot illustrates the parallelism in the usage of different SSOF software releases. Filled circles represent failures, originating in release 1 (or release A) of the software, that were triggered through the next-release effect when the system release shown on the x axis was used.

dence to failure probabilities obtained from formal certification processes. It was concluded that a credible use of software reliability models for prediction of rare events is possible, but that this has to be accompanied by a careful evaluation of the model assumptions and constraints, a validation of the predictive capabilities of the model, and an understanding of what the predictions really mean. [Prue95] noted that while complex software may never be proven to be perfect, the shuttle software has a NASA commitment to be as close as possible.

11.8 Availability

11.8.1 Introduction

An important concept that is closely related to reliability is software availability. The importance stems from prevalent industry specifications related to reliability and availability. For example, one of Bellcore's primary requirements is for the availability of telecommunications network elements. Their target is about 3 minutes of downtime per year. Availability is simply the probability that the system will be available when demanded, and it depends on both the reliability and the reparability of the system. We briefly explore the practical evaluation of system availability by using the following example.

[Cram92] reports on the availability of a large multirelease telecommunications switching system. The system in this study has two principal switching products running similar software, but distinguished by their hardware compositions. We will refer to these separate systems (or products) as P1 and P2. Since the software running on these systems generally changes more frequently than the hardware, a new software release will represent the upgrade to a new system. A software release is normally installed at hundreds of sites. To distinguish among the releases, we use release numbers (R7, R8, etc.). Software libraries for this system exceed 10 million lines of high-level code. A typical executable software load consists of approximately 7 million lines of high-level code, of which about 10 percent is new or modified code. The data used in this example come from the operational phase of the software. The data are collected on a regular basis as part of software process and product quality assurance activities at the organization that developed the system.

11.8.2 Measuring availability

11.8.2.1 Instantaneous availability. Instantaneous availability is the probability that the system will be available at any random time t during its life [Sand63, Triv82]. We estimate instantaneous availability in a period i as follows:

$$\hat{A}(i) = \frac{\text{uptime in period } i}{\text{total in-service time for period } i} \tag{11.8}$$

where the in-service time is the total time in the period i during which all hosts of a particular type (e.g., processor A, processor B), at all sites, operated a particular software release (whether fully operational, partly degraded, or under repair), while uptime is the total time during period i at which the systems *were not* in the "100 percent down" state (or total system outage state).* Correspondingly, the instantaneous unavailability estimate is $(1 - \hat{A}(i))$. Associated with this measure are instantaneous system failure, $\lambda(i)$, and recovery rates, $\rho(i)$, which are estimated as follows:

$$\hat{\lambda}(i) = \frac{\text{number of outages in period } i}{\text{total uptime for period } i} \tag{11.9}$$

$$\hat{\rho}(i) = \frac{\text{number of outages in period } i}{\text{total downtime for period } i} \tag{11.10}$$

* The equation can be customized with other definitions of uptime that suit the need. For example, we could define *uptime* as only the states where the system was 80 to 100 percent operational.

where in-service time for period i is the sum of the downtime and uptime in that period.

11.8.2.2 Average availability. Since the raw data are often noisy, the data are usually presented after some form of smoothing, or data aging, has been applied. This gives rise to a family of smoothed availability metrics (note there is an analogous family of smoothed reliability metrics). Examples are one-sided moving average and symmetrical moving average, such as 11-point symmetrical moving average. An extreme form of smoothing is provided by the *average,* or *uptime,* availability. Uptime availability is the proportion of time in a specified interval $[0,T]$ that the system is available for use [Sand63, Shoo83]. We estimate uptime availability up to and including period i as follows:

$$A_c(i) = \frac{\sum_{x=1}^{i} \text{uptime in period } x}{\sum_{x=1}^{i} \text{total in-service time for period } x} \tag{11.11}$$

Total uptime and total in-service time are cumulative sums starting with the first observation related to a particular release. Uptime includes degraded service. Associated with uptime availability are average system failure, $\hat{\lambda}_c(i)$, and recovery rates, $\hat{\rho}_c(i)$, which are Eq. (11.11) analogues of Eqs. (11.9) and (11.10).

11.8.3 Empirical unavailability

Figure 11.16 illustrates a typical unavailability observed for the system described in [Cram92]. In addition to the instantaneous data, we show the influence of different smoothing approaches. For example, [Cram92] found that the 11-point symmetrical moving-average smoothing was useful for examining trends in the instantaneous unavailability. Each raw data point on the graph corresponds to the data collected during one calendar period.* Abrupt changes in instantaneous unavailability are smoothed by uptime averaging. Under the assumption that product unavailability becomes smaller with time, we would expect uptime unavailability estimates to be generally conservative. Unavailability curves similar to that in Fig. 11.16 were observed for the TROPICO 4096 system described in [Lapr91] (DataSet 4 on the Data Disk; also see Sec. 2.4.4).

* For example, one week or one month. *Note:* In order to draw the raw data on the logarithmic scale, the data points (periods) where no failures were observed (i.e., zero failure rate) were censored and only the adjacent nonzero failure rate points are shown.

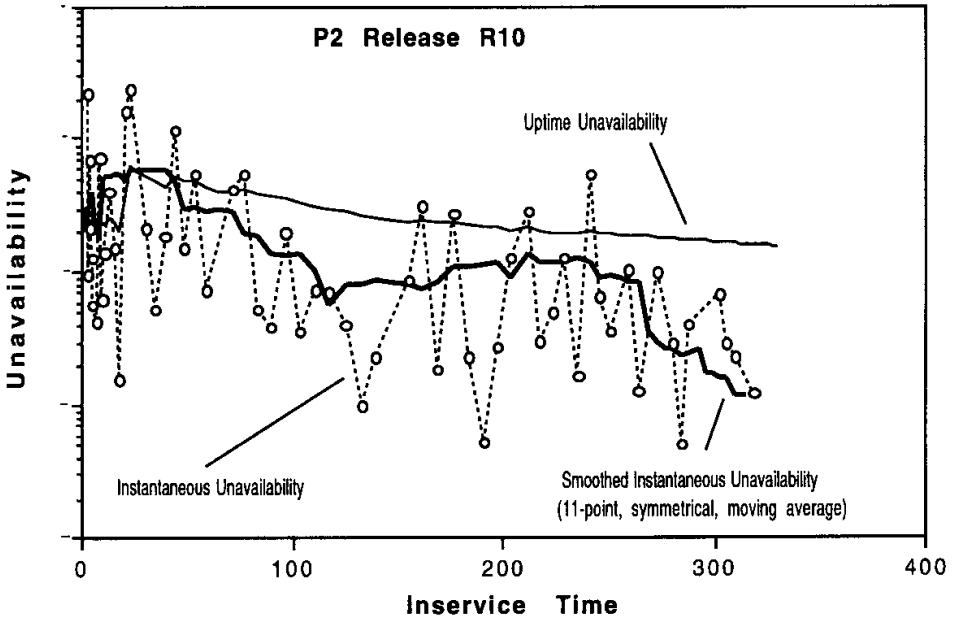

Figure 11.16 Illustration of empirical unavailability data and of the effect of some smoothing options applied to release R10 of P2.

There is often a period, immediately after the product is made available to customers, in which considerable oscillation is observed in unavailability. We refer to the time period from the point where the product is made available to the customers (i.e., time zero) to the point where the instabilities abate as the *transient region.* It corresponds to the transient part of the availability function. It is interesting to note the maximum in the initial part of the smoothed P2 unavailability function is a characteristic of reliability growth [Lapr90b]. In Fig. 11.16 the duration of this region of instability is about 20 to 50 in-service time units, but in general, it depends on the product type and release. Once the instability had decayed, all releases exhibited fairly smooth unavailability decay curves, which in this case could be approximated by almost straight lines [Cram92].

11.8.3.1 Failure and recovery rates. Two measures which directly influence the availability of a system are its failure rate and its field repair rate (or software recovery rate). In a system which improves with field usage we would expect a decreasing function for failure rate with in-service time (implying fault or problem reduction and reliability growth). Failure rate is connected to both the operational usage profile and the process of problem resolution and correction. Measured recovery rate depends on the operational usage profile, the type of problem encountered, and the field response to that problem (i.e., the duration of outages in this case). If the failures encountered during the operational phase of the release do not exhibit durations that would be

preferentially longer or shorter at a point (or period) in the life cycle, then we would expect the instantaneous recovery rate to be a level function with in-service time (with, perhaps, some oscillations in the early stages). This behavior was generally observed in both the [Cram92] and the [Lapr91] investigations. It is interesting to note that [Cram92] observed recovery rates at least 3 to 4 orders of magnitude larger than the failure rate. It was also found that the recovery rate was approximately constant, so the availability was governed primarily by the stochastic changes in the failure rate.

11.8.4 Models

The time-varying nature of both the failure rate and, to a lesser extent, the repair rate indicates that a full availability model should be nonhomogeneous. In addition, the distribution of outage causes, as well as the possibility of operation in degraded states, suggest that a detailed model should be a many-state model. Nonetheless, a very simple two-state model may provide a reasonable description of the system availability beyond the transient region. It can be shown that system availability $A(t)$ and unavailability $\bar{A}(t) = 1 - A(t)$, given some simplifying assumptions, is (e.g., [Triv82, Shoo83]):

$$A(t) = \frac{\rho}{\lambda + \rho} + \frac{\lambda}{\lambda + \rho} e^{-(\lambda + \rho)t} \tag{11.12}$$

It can also be shown that uptime availability can be formulated as

$$A_c(T) = \frac{\rho}{\lambda + \rho} + \frac{\lambda}{(\lambda + \rho)^2 T} - \frac{\lambda}{(\lambda + \rho)^2 T} e^{-(\lambda + \rho)T} \tag{11.13}$$

The system becomes independent of its starting state after operating for enough time for the transient part of the preceding equations to decay away. This steady-state availability of the system is $A(\infty)$ = limit $\{A(t = T \rightarrow \infty)\}$, i.e.,

$$A(\infty) = \frac{\rho}{\lambda + \rho} \tag{11.14}$$

The preceding model represents a two-state system which can be either fully operational or completely off-line and under repair. However, realistic systems, like the ones discussed in our case studies, not only have failure rates and repair rates which vary with time and can have different down states (e.g., FCC-reportable or not [FCC92]), but they can also function in more than one up state (i.e., the system may remain operational but with less than 100 percent functionality for

some failures). Thus, a many-state nonhomogeneous Markov model may be more appropriate for describing the details of such systems (e.g., [Triv75, Lapr92b, Ibe92]). Nevertheless, a classical two-state model for availability of recoverable systems can be used to approximate behavior of more complex nonhomogeneous systems.

11.8.4.1 Practical approximations. An approximation that may work quite well is what we will call the *steady-state approximation.* It is based on the observations made by [Triv75, Shoo83]. We note that once the system has been operational for some time, the steady-state Eq. (11.14) may be used to approximate the instantaneous availability by assuming a piecewise-constant variation of λ and ρ in time. Letting $\hat{\lambda}(t)$ and $\hat{\rho}(t)$ be estimates at time t for λ and failure rate, ρ, respectively, we can estimate instantaneous availability as

$$\hat{A}(t) = \frac{\hat{\rho}(t)}{\hat{\lambda}(t) + \hat{\rho}(t)} \tag{11.15}$$

The $\hat{\lambda}(t)$ and $\hat{\rho}(t)$ approximations can be obtained from the empirical data: the former through application of a reliability model and the latter is often assumed to be a constant (e.g., [Lapr91]).

The following example provides a simple illustration of the application of the preceding approximation. We consider the release R11 for product P2. From other work [Cram92] we know that the uptime recovery rate for this release is approximately constant once sufficient in-service time has passed. We make the simplifying assumption that the recovery rate is constant and choose it to be the average of the period being considered (i.e., it is the uptime recovery rate of the sample point with the largest in-service time). Furthermore, the MO model provides a good descriptive, as well as predictive, model for the failure rate of the P1 and P2 systems [Jone91, Jone92].

The MO failure rate equation with the parameters obtained from a fit and constant repair rate were used to compute the steady-state approximations for instantaneous unavailability. The results are shown Fig. 11.17. The figure also shows the result of another, more accurate, computational approximation we call *n-step approximation.* The n-step transition approximation [Cram92] is based on numerical solution of Chapman-Kolmogorov equations that describe the system states.

It is interesting to note that the n-step approximation reflects the transient region maximum expected in the unavailability function of a system that experiences reliability growth [Lapr92b], while the steady-state approximation does not exhibit this mode. However, the results from the n-step and steady-state approximations practically coincide

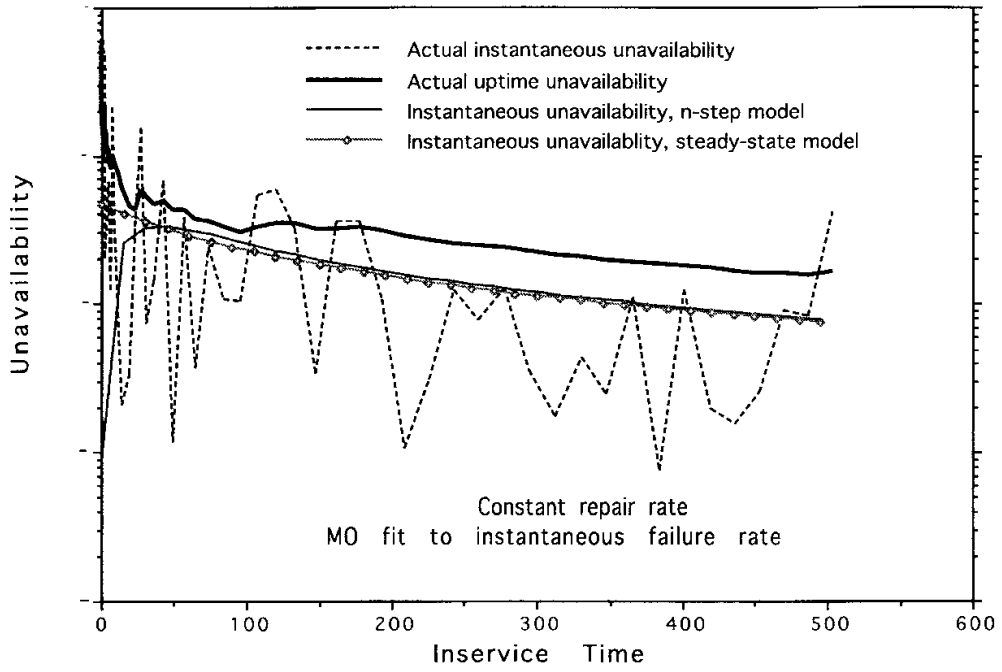

Figure 11.17 Unavailability modeling using the MO failure intensity model and constant repair rate. Both the MO parameters and the average repair rate were estimated on the basis of *all* data. The data are for system P2, release R11.

once past the transient period, and for all practical purposes, the computationally simpler steady-state approximation may be all that is needed to model availability of a system. Of course, both approximations lie below uptime unavailability (shown in the figure as a thick solid line) because the instantaneous unavailability is less conservative than the uptime unavailability.

11.8.4.2 Prediction. In practice, a model would be used to predict future unavailability of a system. Of course, only the data up to the point from which the prediction is being made would be available. In this example, we refer to the point at which the prediction is made as the data *cut-off point.* The prediction of future unavailability will differ from the true value depending on how well the model describes the system.

In Fig. 11.18, we show instantaneous and uptime unavailability using the steady-state approximation, average recovery rate at the cut-off point, and the MO failure fit to points from the beginning of the operational phase of the release up to the cut-off point. We see that the approximation for uptime unavailability appears to follow the empirical uptime unavailability quite well. Similar results have been obtained for other releases of the same product. Of course, in practice, predictive characteristics of a model should be checked more formally using tests such as u-plots or relative error calculations (see Sec. 11.3.3.2). However, the lesson is that, from a practical perspective, a

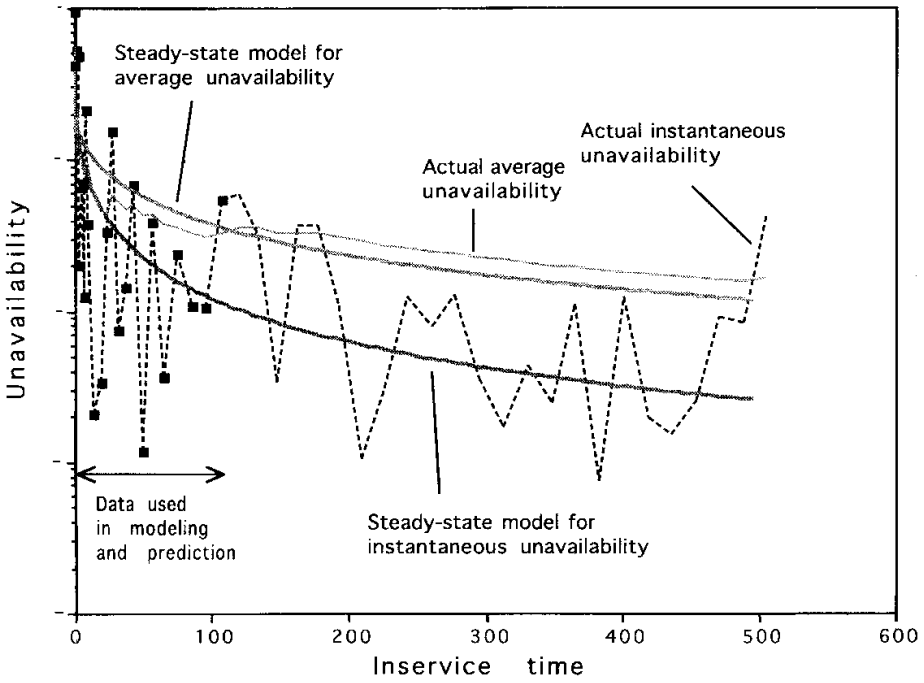

Figure 11.18 Unavailability was modeled using the steady-state model. Failure intensity was modeled using MO model and average repair rate. Both the MO parameters and the average repair rate were estimated on the basis of the first 100 in-service time units. The data are for system P2 release R11.

relatively simple availability model can have quite reasonable predictive properties for a system that is being released and maintained in a stable environment.

Availability models can be constructed using other assumptions, including time-varying repair rate, and different (appropriate) software reliability models. An example is the availability model constructed in [Lapr91] using the constant-repair-rate assumption, and the hyperexponential reliability model to fit the empirical failure data for the TROPICO-R 4096 switch software. The model is based on calendar-time usage, rather than on in-service-time usage, and it has good predictive characteristics (see Sec. 2.4.4).

11.9 Summary

Analysis of field data is an extremely important software reliability engineering activity. It enables quantitative and qualitative assessment of the product quality during its actual usage; it provides the link between software quality in the field and software life-cycle processes; and it provides the basis for active control of software field quality. It is essential that you perform the reliability analysis on properly collected data, using adequate tools and appropriate software reliability models. The key data collection issues are (1) consistent definition of failures

and faults, (2) measurement of the product usage, and (3) the data granularity. Analysis tools and methods include graphs and plots. In that context, it is interesting to note that an approach that can be very useful, but is often overlooked in practice, is reliability model selection based on graphical diagnostics. The customer's usage of the product is very important to understanding of the current or future reliability of a system. If CPU-time-usage data are difficult to obtain (as is often the case), field failure rates based on calendar time can be employed. Experience shows that both calendar-time- and usage-time-based models can be used effectively to predict system behavior. In the case of critical systems, and in situations where we are dealing with rare failure events, common software reliability models may not be adequate and special methods need to be used. A concept that is closely related to reliability, and has many practical implications, is system availability. In addition to failure data, availability analysis requires collection of information about corresponding system recovery rates.

Problems

11.1 Outline a study document for investigation of the field reliability of your favorite software system. Include explicit study goals, a template for data collection and an outline of the processes and methodology you intend to use to validate your data, and analyze and present the results.

11.2 *a.* List at least five important failure data collection issues.
b. Explain the statement: "Ten usage weeks may correspond to three calendar days."

11.3 What is the difference between exploratory and confirmatory data analyses and techniques?

11.4 What are the advantages and disadvantages of using plots, graphs, and graphical model selection and diagnostic tools? Explain and give examples.

11.5 Using DataSet 1 (large telecommunications system):
a. Analyze the usage and office data and comment on any anomalies. Are they explainable?
b. Calculate the failure intensity using failure grouping of 10 percent and plot the results.
c. Compare the plot with Figs. 11.4 and 11.7 and comment on the difference, if any.

11.6 Perform Laplace trend analysis of the DataSet 1 using one of the automated tools (e.g., SoRel, described in Sec. A.7).

11.7 Using DataSet 1 (large telecommunications system):

a. Calculate the maximum likelihood estimates for the parameters of the MO model based on all available data.
b. Calculate the maximum likelihood estimates for the parameters of the MO model at 10 percent time intervals and forecast the end time. Compute the relative error of the prediction at each time point and create a relative error plot using the maximum likelihood estimates (MLE).
c. Choose a model other than the log Poisson and repeat item *b.* Is the new model better or worse? Explain and justify using a metric such as u-plot.

11.8 *a.* Use DataSet 1 from the Data Disk and replot Fig. 11.8.
b. Can we analyze field data using usage-time-based reliability models in the calendar-time domain? Explain, giving examples.

11.9 What is the relationship between the field usage of the software and the number of sites at which the product is installed? Using information in DataSet 1 and Fig. 11.8, develop an analytic form for this relationship.

11.10 What is the difference between failures or outages per in-service time, and failures and outages per system per calendar year? If the product is a telephone switching software, which one would you use to examine quality of service offered to a typical telephone user? Explain.

11.11 What are the advantages and disadvantages of using S-shaped reliability models for modeling field data? Explain and give examples.

11.12 Perform an exploratory analysis of the DataSet 2 (IBM Corporation *controller software*).
a. Group the data into categories one month long and compute the corresponding failure frequencies; plot the frequency against the calendar time.
b. Apply symmetrical 3-month and 5-month moving-average smoothing to the frequency data and overlay the results on the frequency plot from item *a.* How do results change if the moving average is asymmetrical (only the older data are used to adjust the current value)?
c. For each failure, calculate its corresponding time to failure. Tabulate the results.
d. Compute calendar-time failure intensity using failure groups of size 5; plot the failure intensity against the calendar time.
e. Compute and plot Laplace trend for the data set.
f. Try to linearize the data.
g. Plot cumulative failure distribution against calendar time.
h. Discuss your results.

11.13 *a.* Based on Prob. 11.12 results, recommend a model.
b. Fit the model to the data, record the parameters, and overlay the plot of the modeled and empirical failure intensity, and modeled and empirical cumulative failure distributions.

c. Discuss the validity and predictive capabilities of the model by showing how well it predicts the total cumulative failure count in month 36, given data up to month 12, month 18, month 24, and month 30.
d. Use a computational diagnostic to evaluate the predictions (justify the use of this particular diagnostic).

11.14 What is the next-release effect? Explain and give examples.

11.15 Perform full reliability analysis of DataSet 3.

11.16 Fit and justify a model for DataSet 5. What can we expect the field failure ratio to be in 1996? Provide upper and lower bounds for this estimate.

11.17 Look up u-plots and prequential likelihood tests. Apply these methods to one of the analyses requested in the previous problems. Explain and justify your approach and the results.

11.18 A software system has a failure intensity objective of 0.005 failures per CPU hour. During beta test the system runs for 700 CPU hours without a failure.

a. What is the upper 95 percent confidence bound on the program failure intensity?
b. If the upper 99 percent confidence bound on the failure intensity must be below 0.005, how many *more* CPU hours would you have to beta-test the system without experiencing a failure.

11.19 a. Why is it that time to next failure may be totally inapplicable to field data?
b. Perform an exploratory analysis of DataSet 6, including next-release effect and parallelism analyses. Use dot plots. Write an exploratory analysis report on SSOF.
c. Is the Weibull defect model a viable model for SSOF? Discuss, using examples and analyses based on DataSet 6.

11.20 Describe a simple two-state Markov chain model for availability and derive the basic relationships for instantaneous and long-term (steady-state) availability.

11.21 a. Use Shooman's steady-state approximation model to analyze availability for TROPICO R-4096 switching software given in DataSet 4.
b. Compare the results with the hyperexponential model.