Part

1

Technical Foundations

Chapter 1

Introduction

Michael R. Lyu
AT&T Bell Laboratories

1.1 The Need for Reliable Software

With the advent of the electronic digital computer 50 years ago [Burk46], we have become dependent on computers in our daily lives. The computer revolution is fueled by an ever more rapid technological advancement. Today, computer hardware and software permeates our modern society. Computers are embedded in wristwatches, telephones, home appliances, buildings, automobiles, and aircraft. Science and technology demand high-performance hardware and high-quality software for making improvements and breakthroughs. We can look at virtually any industry—automotive, avionics, oil, telecommunications, banking, semiconductors, pharmaceuticals—all these industries are highly dependent on computers for their basic functioning.

The size and complexity of computer-intensive systems has grown dramatically during the past decade, and the trend will certainly continue in the future. Contemporary examples of highly complex hardware/software systems can be found in projects undertaken by NASA, the Department of Defense, the Federal Aviation Administration, the telecommunications industry, and a variety of other industries. For instance, the NASA Space Shuttle flies with approximately 500,000 lines of software code on board and approximately 3.5 million lines of code in ground control and processing. After being scaled down significantly from its original plan, the International Space Station Alpha is still projected to have millions of lines of software to operate innumerable hardware pieces for its navigation, communication, and experimentation. In the telecommunications industry, operations for phone carriers are supported by hundreds of software systems, with hundreds of millions of lines of source code. In the avionics industry, almost all new payload instruments contain their own microprocessor system with extensive embedded software. A massive amount of hardware and complicated software also exists in the Federal Aviation Administration's Advanced Automation System, the new generation air traffic control system. In our offices and homes, personal computers cannot function without complex operating systems (e.g., Windows) ranging from 1 to 5 million lines of code, and many other shrink-wrapped software packages of similar size provide a variety of applications for our daily use of these computers.

The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them. When the requirements for and dependencies on computers increase, the possibility of crises from computer failures also increases. The impact of these failures ranges from inconvenience (e.g., malfunctions of home appliances) to economic damage (e.g., interruptions of banking systems) to loss of life (e.g., failures of flight systems or medical software). Needless to say, the reliability of computer systems has become a major concern for our society.

Within the computer revolution, progress has been uneven: software assumes a larger burden, while based on a less firm foundation, than hardware. It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope, and it is the growth in utilization of software components that is largely responsible for the high overall complexity of many system designs. However, in stark contrast with the rapid advancement of hardware technology, proper development of software technology has failed to keep pace in all measures, including quality, productivity, cost, and performance. With the last decade of the 20th century, computer software has already become the major source of reported outages in many systems [Gray90]. Consequently, recent literature is replete with horror stories of projects gone awry, generally as a result of problems traced to software.

Software failures have impaired several high-visibility programs. In the NASA Voyager project, the Uranus encounter was in jeopardy because of late software deliveries and reduced capability in the Deep Space Network. Several Space Shuttle missions have been delayed due to hardware/software interaction problems. In one DoD project, software problems caused the first flight of the AFTI/F-16 jet fighter to be delayed over a year, and none of the advanced modes originally planned could be used. Critical software failures have also affected numerous civil and scientific applications. The ozone hole over Antarctica would have received attention sooner from the scientific community if a data analysis program had not suppressed the anomalous data because it was "out of range." Software glitches in an automated baggage-handling system forced Denver International Airport to sit empty more than a year after airplanes were to fill its gates and runways [Gibb94].

Unfortunately, software can also kill people. The massive Therac-25 radiation therapy machine had enjoyed a perfect safety record until software errors in its sophisticated control systems malfunctioned and claimed several patients' lives in 1985 and 1986 [Lee92]. On October 26, 1992, the Computer Aided Dispatch system of the London Ambulance Service broke down right after its installation, paralyzing the capability of the world's largest ambulance service, which handles 5000 daily requests to transport patients in emergency situations [SWTR93]. In the highly automated aviation industry, misunderstandings between computers and pilots have been implicated in several airline crashes in the past few years [Swee95], and in some cases experts hold software control responsible because of inappropriate reaction of the aircraft to the pilots' desperate inquiries during an abnormal flight.

Software failures also have led to serious consequences in business. On January 15, 1990, a fault in a switching system's newly released software caused massive disruption of a major carrier's long-distance network, and another series of local phone outages traced to software problems occurred during the summer of 1991 [Lee92]. These critical failures caused enormous revenue losses to thousands of companies relying on telecommunications companies to support their businesses.

Many software systems and packages are distributed and installed in identical or similar copies, all of which are vulnerable to the same software failure. This is why even the most powerful software companies such as Microsoft are fearful of "killer bugs" which can easily wipe out all the profits of a glorious product if a recall is required on the tens of millions of copies they have sold [Cusu95]. To this end, many software companies see a major share of project development costs identified with the design, implementation, and assurance of reliable software, and they recognize a tremendous need for systematic approaches using *software reliability engineering* techniques. Clearly, developing the required techniques for software reliability engineering is a major challenge to computer engineers, software engineers, and engineers of various disciplines now and for decades to come.

1.2 Software Reliability Engineering Concepts

Software reliability engineering is centered around a very important software attribute: *reliability*. Software reliability is defined as *the probability of failure-free software operation for a specified period of time in a specified environment* [ANSI91]. It is one of the attributes of software quality, a multidimensional property including other customer satisfaction factors such as functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation [Grad87, Grad92]. Software reliability, however, is generally accepted as the key factor in software quality since it quantifies software failures—which can make a powerful system inoperative or, as with the Therac-25, deadly. As a result, reliability is an essential ingredient in customer satisfaction. In fact, ISO 9000-3 specifies measurement of field failures as the only required quality metric: "... at a minimum, some metrics should be used which represent reported field failures and/or defects from the customer's viewpoint.... The supplier of software products should collect and act on quantitative measures of the quality of these software products." (See sec. 6.4.1 of [ISO91].)

Example 1.1 shows the impact of high-severity failures to customer satisfaction.

Example 1.1 A survey of nine large software projects was taken in [Merc94] to study the factors contributing to customer satisfaction. These projects were telecommunications systems responsible for day-to-day operations in the U.S. local telephone business. The survey requested telephone customers to assess a quality score between 0 and 100 for each system. The average size of these projects was 1 million lines of source code.

Trouble Reports (i.e., failure reports in the field) were collected from these projects. Figure 1.1 shows the overall quality score from the survey of these projects, plotted against the number of high-severity Trouble Reports.

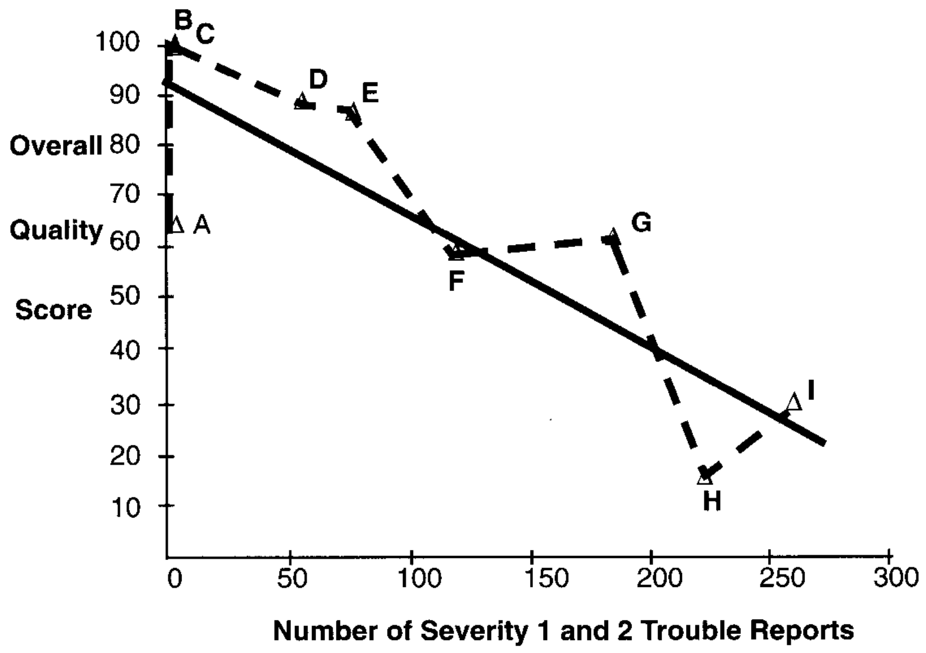

Figure 1.1 Correlations between software quality and high-severity failures.

From Fig. 1.1 we can observe a high negative correlation (–0.86) between the overall quality score and the number of high-severity failures for each project. This example illustrates that in the telecommunications industry, the number of critical software failures promptly indicates negative customer perception on overall software quality. This quality indicator is also generally applicable to many other industries.

Software reliability is also one of the system dependability concepts that are discussed in detail in Chap. 2. Example 1.2 demonstrates the impact of software reliability to system reliability.

Example 1.2 A military distributed processing system has an MTTF (mean time to failure, see definition in Sec. 1.4) requirement of 100 hours and an availability requirement of 0.99. The overall architecture of the system is shown in Fig. 1.2, indicating that the system consists of three subsystems, SYS1, SYS2, SYS3, a local area network, LAN, and a 10-kW power generator GEN. In order for the system to work, all the components (except SYS2) have to work. In the early phase of system testing, hardware reliability parameters are predicted according to the MIL-HDBK-217 and shown for each system component. Namely, above each component block in Fig. 1.2, two numbers appear. The upper number represents the predicted MTTF for that component, and the lower number represents its MTTR (mean time to repair, see Sec. 1.4). The units are hours. For example, SYS1 has 280 hours for MTTF and 0.53 hours for MTTR, while SYS2 and SYS3 have 387 hours for MTTF and 0.50 hours for MTTR. Note that SYS2 is configured as a triple-module redundant system, shown in the dotted-line block, where the subsystem will work as long as two or more modules work. Due to this fault-tolerant capability, its MTTF improves to 5.01 × 10^4 hours and MTTR becomes 0.25 hours.

To calculate the overall system reliability, all the components in the system have to be considered. If we assume the software does not fail (a mistake often made by system reliability engineers!), the resulting system MTTF would be 125.9

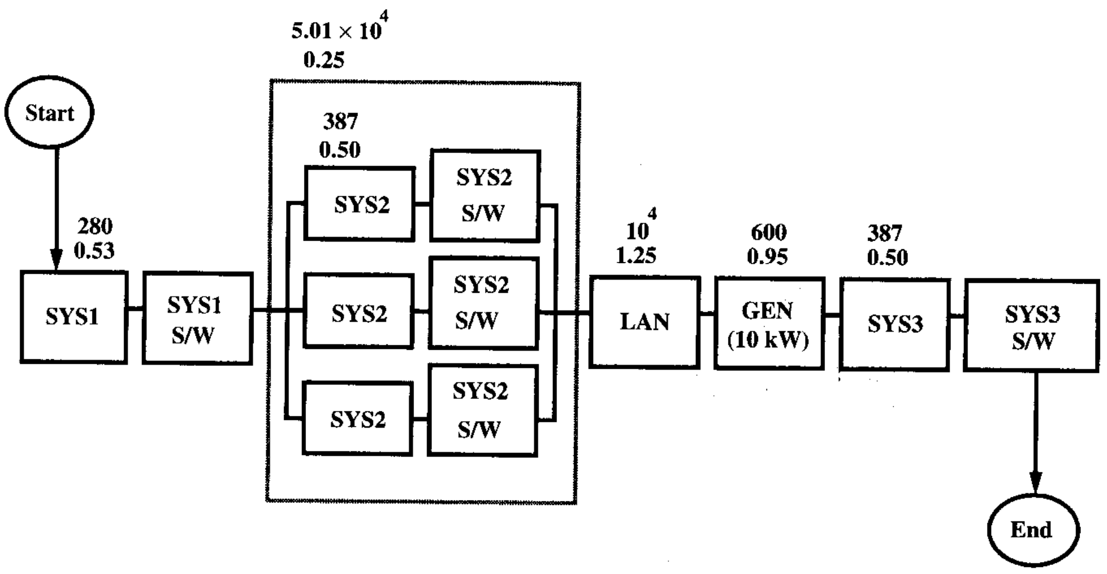

Figure 1.2 An example of predicting system reliability.

hours and MTTR would be 0.62 hours, achieving system availability of 0.995. It looks as if the system already meets its original requirements.

But the software does fail. Both SYS2 and SYS3 software contain 300,000 lines of source code, and following the prediction model described in Chap. 3 (Sec. 3.8.3) and [RADC87], the predicted initial failure rates for SYS2 and SYS3 software are both 2.52 failures per execution hour. (Note the three SYS2 S/W are identical software copies and not fault-tolerant.) Even without considering SYS1 software failures, the system MTTF would have become 11.9 CPU minutes! Assuming MTTR is still 0.62 hours (although it should be higher since it generally takes longer to reinitialize the software) and CPU time and calendar time are close to each other (which is true for this distribution system), the system availability becomes 0.24—far less than predicted!

Note that the system presented in Example 1.2 was a real-world example, and the estimated reliability parameters were actual practices following military handbooks [Lyu89]. This example is not an extreme case. In fact, many existing large systems face the same situation: software reliability is the bottleneck of system reliability, and the maturity of software always lags behind that of hardware. Accurately modeling software reliability and predicting its trend have become critical, since this effort provides crucial information for decision making and reliability engineering for most projects.

Reliability engineering is a daily practiced technique in many engineering disciplines. Civil engineers use it to build bridges and computer hardware engineers use it to design chips and computers. Using a similar concept in these disciplines, we define *software reliability engineering* (SRE) as *the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability* [IEEE95]. SRE therefore includes:

1. Software reliability measurement, which includes estimation and prediction, with the help of software reliability models established in the literature
2. The attributes and metrics of product design, development process, system architecture, software operational environment, and their implications on reliability
3. The application of this knowledge in specifying and guiding system software architecture, development, testing, acquisition, use, and maintenance

Based on the above definitions, this book details current SRE techniques and practices.

1.3 Book Overview

Mature engineering fields classify and organize proven solutions in handbooks so that most engineers can consistently handle complicated but routine designs. Unfortunately, handbooks of software engineering practice are unknown. Software development has been treated as an art. Although we understand a very large part of this art, it is still not a practiced engineering discipline. Consequently, mistakes in software development are repeated project after project, year after year, and the software crises of 25 years ago are still with us today [Gibb94].

Fortunately, the reliability component of software engineering is evolving from an art to a practical engineering discipline. It is time to begin to codify our knowledge in SRE and make it available—this is the main purpose of this handbook. This handbook provides information on the key methods and methodologies used in SRE, covering its state-of-the-art techniques and state-of-practice approaches. The book is divided into three parts and 17 chapters. Each chapter is written by SRE experts, including researchers and practitioners. These chapters cover the theory, design, methodology, modeling, evaluation, experience, and assessment of SRE techniques and applications.

Part 1 of the book, composed of five chapters, sets up the *technical foundations* for software reliability modeling techniques, in which system-level dependability and reliability concepts, software reliability prediction and estimation models, model evaluation and recalibration techniques, and operational profile techniques are presented. In particular,

1. Chapter 1 gives an *introduction* of the book, where its framework is outlined and the main contents of each chapter are surveyed. Basic ideas, terminology, and techniques in SRE are presented.
2. Chapter 2 provides a general overview of the *system dependability* concept, and shows that the classical reliability theory can be extended in order to be interpreted from both hardware and software viewpoint.
3. Chapter 3 reviews the major software *reliability models* that appear in the literature, from both historical and applications perspectives. Each model is presented with its motivation, model assumptions, data requirements, model form, estimation procedure, and general comments about its usage.
4. Chapter 4 presents a systematic framework to conduct *model evaluation* of several competing reliability models, using advanced statistical criteria. Recalibration techniques which can greatly improve model performance are also introduced.
5. Chapter 5 details a technique that is essential to SRE: the *operational profile.* The operational profile shows you how to increase productivity and reliability and speed development by allocating project resources to functions on the basis of how a system will be used.

Part 2 contains SRE *practices and experiences* in six chapters. This part of the book consists of practical experiences from major organizations such as AT&T, JPL, Bellcore, Tandem, IBM, NASA, Nortel, ALCATEL, and other international organizations. Various SRE procedures are implemented for particular requirements under different environments. The authors of each chapter in Part 2 describe the practical procedures that work for them, and convey to you their experiences and lessons learned. Specifically,

1. Chapter 6 describes the best *current practice* in SRE adopted by over 70 projects at AT&T. This practice allows you to analyze, manage, and improve the reliability of software products, to balance customer needs in terms of cost, schedule, and quality, and to minimize the risks of releasing software with serious problems.
2. Chapter 7 conveys the *measurement experience* in applying software reliability models to several large-scale projects at JPL and Bellcore. We discuss the SRE procedures, data collection efforts, modeling approaches, data analysis methods, reliability measurement results, lessons learned, and future directions. A practical scheme to improve measurement accuracy by linear combination models is also presented.
3. Chapter 8 shows *measurement-based analysis* techniques which directly measure software reliability through monitoring and recording failure occurrences in a running system under various user workloads. Experiences with Tandem GUARDIAN, IBM MVS, and VAX VMS operating systems are explored.
4. Chapter 9 proposes a *defect classification* scheme which extracts semantic information from software defects such that it provides a measurement on the software development process. This chapter explains the framework, procedure, and advantage of this scheme and its successful application and deployment in many projects at IBM.
5. Chapter 10 addresses software reliability *trend analysis,* which can help project managers control the progress of the development activities and determine the efficiency of the test programs. Application results from a number of studies including switching systems and avionic applications are reported.
6. Chapter 11 provides insight into the process of collecting and analyzing software reliability *field data* through a discussion of the underlying principles and case study illustrations. Included in the field data analysis are projects from IBM, Hitachi, Nortel, and space shuttle onboard flight software.

Emerging techniques which have been used to advance SRE research field are addressed by the six chapters in Part 3. These techniques include software metrics, testing schemes, fault-tolerant software, fault tree analysis, simulation, and neural networks. After explicitly explaining these techniques in concrete terms, authors of the chapters in Part 3 establish the relationships between these techniques and software reliability. Potential research topics and their directions are also addressed in detail. In summary,

1. Chapter 12 presents the technique to incorporate software *metrics* for reliability assessment. This chapter makes the connection between software complexity and software reliability, in which both functional complexity and operational complexity of a program are examined for the development and maintenance of reliable software.
2. Chapter 13 explores the relationship between software *testing* and reliability. In addressing the impact of testing to reliability, this chapter applies program structure metrics and code coverage data for the estimation of software reliability and the assessment of the risk associated with software.
3. Chapter 14 focuses on the software *fault tolerance* approach as a potential technique to improve software reliability. Issues regarding the architecture, design, implementation, modeling, failure behavior, and cost of fault tolerant systems are discussed.
4. Chapter 15 introduces the *fault tree* technique for the reliability analysis of software systems. This technique helps you to analyze the impact of software failures on a system, to combine off-line and on-line tests to prevent or detect software failures, and to compare different design alternatives for fault tolerance with respect to both reliability and safety.
5. Chapter 16 demonstrates how several *simulation* techniques can be applied to a typical software reliability engineering process, in which many simplifying assumptions in reliability modeling could be lifted. This chapter shows the power, flexibility, and potential benefits that the simulation techniques offer, together with methods for representing artifacts, activities, and events of the reliability process.
6. Chapter 17 elaborates how the *neural networks* technology can be used in software reliability engineering applications, including its usage as a general reliability growth model for better predictive accuracy, and its exercise as a classifier to identify fault-prone software modules.

In addition to these book chapters, two appendixes and an MS/DOS diskette are enclosed in the book. Appendix A surveys the currently available tools which encapsulate software reliability models and techniques. These tools include AT&T Toolkit, SMERFS, SRMP, SoRel, CASRE, and SoftRel. Appendix B reviews the analytical modeling techniques, statistical techniques, and reliability theory commonly used in the SRE studies. The MS/DOS disk, called *Data and Tool Disk* (or *Data Disk*), includes two directories: the DATA directory and the TOOL directory. The DATA directory contains more than 40 published and unpublished software failure data sets used in the book chapters, and the TOOL directory contains the AT&T SRE Toolkit, SMERFS, CASRE, and SoftRel software reliability tools.

Finally, at the end of each book chapter are problems which provide practice exercises for the reader.

1.4 Basic Definitions

We notice three major components in the definition of software reliability: *failure, time,* and *operational environment.* We now define these terms and other related SRE terminology. We begin with the notions of a software system and its expected service.

Software systems. A *software system* is an interacting set of software subsystems that is embedded in a computing environment that provides inputs to the software system and accepts service (outputs) from the software. A software subsystem itself is composed of other subsystems, and so on, to a desired level of decomposition into the smallest meaningful elements (e.g., modules or files).

Service. Expected *service* (or "behavior") of a software system is a time-dependent sequence of outputs that agrees with the initial specification from which the software implementation has been derived (for the verification purpose), or which agrees with what system users have perceived the correct values to be (for the validation purpose).

Now we observe the following situation: a software system named *program* is delivering an *expected service* to an environment or a person named *user.*

Failures. A *failure* occurs when the user perceives that the program ceases to deliver the expected service.

The user may choose to identify several *severity* levels of failures, such as: catastrophic, major, and minor, depending on their impacts to the system service. The definitions of these severity levels vary from system to system.

Outages. An *outage* is a special case of a failure that is defined as a loss or degradation of service to a customer for a period of time (called *outage duration*). In general, outages can be caused by hardware or software failures, human errors, and environmental variables (e.g., lightning, power failures, fire). A failure resulting in the loss of functionality of the entire system is called a *system outage*. An example to quantify a system outage in the telecommunications industry is to define the outage duration of telephone switching systems to be "greater than 3 seconds (due to failures that results in loss of stable calls) or greater than 30 seconds (for failures that do not result in loss of stable calls)." [BELL90c]

Faults. A fault is uncovered when either a failure of the program occurs or an internal error (e.g., an incorrect state) is detected within the program. The cause of the failure or the internal error is said to be a *fault*. It is also referred as a *bug*.

In most cases the fault can be identified and removed; in some cases it remains a hypothesis that cannot be adequately verified (e.g., timing faults in distributed systems).

In summary, a software failure is an incorrect result with respect to the specification or an unexpected software behavior perceived by the user at the boundary of the software system, while a software fault is the identified or hypothesized cause of the software failure.

Defects. When the distinction between fault and failure is not critical, *defect* can be used as a generic term to refer to either a fault (cause) or a failure (effect). Chapter 9 provides a complete and practical classification of software defects from various perspectives.

Errors. The term *error* has two different meanings:

1. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. Errors occur when some part of the computer software produces an undesired state. Examples include exceptional conditions raised by the activation of existing software faults, and incorrect computer status due to an unexpected external interference. This term is especially useful in fault-tolerant computing to describe an intermediate stage in between faults and failures.
2. A human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. However, this is not a preferred usage, and the term *mistake* is used instead to avoid the confusion.

Time. Reliability quantities are defined with respect to time, although it is possible to define them with respect to other bases such as program runs. We are concerned with three types of time: the *execution time* for a software system is the CPU time that is actually spent by the computer in executing the software; the *calendar time* is the time people normally experience in terms of years, months, weeks, days, etc.; and the *clock time* is the elapsed time from start to end of computer execution in running the software. In measuring clock time, the periods during which the computer is shut down are not counted.

It is generally accepted that execution time is more adequate than calendar time for software reliability measurement and modeling. However, reliability quantities must ultimately be related back to calendar time for easy human interpretation, particularly when managers, engineers, and customers want to compare them across different systems. As a result, translations between calendar time and execution time are required. The technique for such translations is described in [Musa87]. If execution time is not readily available, approximations such as clock time, weighted clock time, staff working time, or units that are natural to the application (such as transactions or test cases executed) may be used.

Failure functions. When a time basis is determined, failures can be expressed in several ways: the cumulative failure function, the failure intensity function, the failure rate function, and the mean time to failure function. The *cumulative failure function* (also called the *mean value function*) denotes the average cumulative failures associated with each point of time. The *failure intensity function* represents the rate of change of the cumulative failure function. The *failure rate function* (also called the *rate of occurrence of failures*) is defined as the probability that a failure per unit time occurs in the interval $[t, t + \Delta t]$, given that a failure has not occurred before t. The *mean time to failure* (MTTF) function represents the expected time that the next failure will be observed. (MTTF is also known as MTBF, mean time between failures.) Note that the above four measures are closely related and could be transposed with one another. Appendix B provides the mathematics of these functions in detail.

Mean time to repair and availability. Another quantity related to time is *mean time to repair* (MTTR), which represents the expected time until a system will be repaired after a failure is observed. When the MTTF and MTTR for a system are measured, its availability can be obtained. *Availability* is the probability that a system is available when needed. Typically, it is measured by

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Chapter 2 (Sec. 2.4.4) gives a theoretical model for availability, while Chap. 11 (Sec. 11.8) provides some practical examples of this measure.

Operational profile. The *operational profile* of a system is defined as the set of operations that the software can execute along with the probability with which they will occur. An operation is a group of runs which typically involve similar processing. A sample operational profile is illustrated in Fig. 1.3. Note that, without loss of generality, the operations can be located on the x axis in order of the probabilities of their occurrence.

Chapter 5 provides a detailed description on the structure, development, illustration, and project application of the operational profile. In general, the number of possible software operations is quite large. When it is not practical to determine all the operations and their probabilities in complete detail, operations based on grouping or partitioning of input states (or system states) into domains are determined. In the situations where an operational profile is not available or only an approximation can be obtained, you may make use of code coverage data generated during reliability growth testing to obtain reliability estimates. Chapter 13 describes some methods for doing so.

Failure data collection. Two types of failure data, namely *failure-count data* and *time-between-failures data,* can be collected for the purpose of software reliability measurement.

Failure-count (or failures per time period) data. This type of data tracks the number of failures detected per unit of time. Typical failure-count data are shown in Table 1.1.

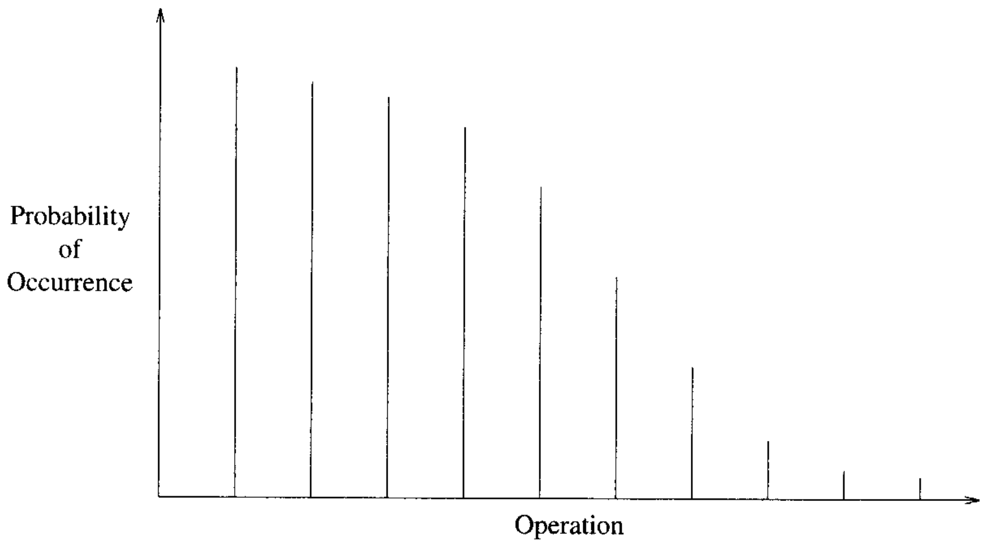

Figure 1.3 Operational profile.

TABLE 1.1 Failure-Count Data

Time (hours)	Failures in the period	Cumulative failures
8	4	4
16	4	8
24	3	11
32	5	16
40	3	19
48	2	21
56	1	22
64	1	23
72	1	24

Time-between-failures (or interfailure times) data. This type of data tracks the intervals between consecutive failures. Typical time-between-failures data can be seen in Table 1.2.

Many reliability modeling programs have the capability to estimate model parameters from either failure-count or time-between-failures data, as statistical modeling techniques can be applied to both. However, if a program accommodates only one type of data, it may be required to transform the other type.

TABLE 1.2 Time-Between-Failures Data

Failure number	Failure interval (hours)	Failure times (hours)
1	0.5	0.5
2	1.2	1.7
3	2.8	4.5
4	2.7	7.2
5	2.8	10.0
6	3.0	13.0
7	1.8	14.8
8	0.9	15.7
9	1.4	17.1
10	3.5	20.6
11	3.4	24.0
12	1.2	25.2
13	0.9	26.1
14	1.7	27.8
15	1.4	29.2
16	2.7	31.9
17	3.2	35.1
18	2.5	37.6
19	2.0	39.6
20	4.5	44.1
21	3.5	47.6
22	5.2	52.8
23	7.2	60.0
24	10.7	70.7

Transformations between data types. If the expected input is failure-count data, it may be obtained by transforming time-between-failures data to cumulative failure times and then simply counting the number of failures whose cumulative times occur within a specified time period. If the expected input is time-between-failures data, converting the failure-count data can be achieved by either *randomly* or *uniformly* allocating the failures for the specified time intervals, and then by calculating the time periods between adjacent failures. Some software reliability tools surveyed in App. A (e.g., SMERFS and CASRE) incorporate the capability to perform these data transformations.

Software reliability measurement. Measurement of software reliability includes two types of activities: reliability *estimation* and reliability *prediction.*

Estimation. This activity determines *current* software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability and determine whether a reliability model is a good fit in retrospect.

Prediction. This activity determines *future* software reliability based upon available software metrics and measures. Depending on the software devêlopment stage, prediction involves different techniques:

1. When failure data are available (e.g., software is in system test or operation stage), the estimation techniques can be used to parameterize and verify software reliability models, which can perform future reliability prediction.
2. When failure data are not available (e.g., software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to determine reliability of the software upon testing or delivery.

The first definition is also referred to as *reliability prediction* and the second definition as *early prediction.* When there is no ambiguity in the text, only the word *prediction* will be used.

Most current software reliability models fall in the estimation category to do reliability prediction. Nevertheless, a few early prediction models were proposed and described in the literature. A survey of existing estimation models and some early prediction models can be found in Chap. 3. Chapter 12 provides some product complexity metrics which can be used for early prediction purposes.

Software reliability models. A software reliability *model* specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment. Figure 1.4 shows the basic ideas of software reliability modeling.

In Fig. 1.4, the failure rate of a software system is generally decreasing due to the discovery and removal of software failures. At any particular time (say, the point marked "present time"), it is possible to observe a history of the failure rate of the software. Software reliability modeling forecasts the curve of the failure rate by statistical evidence. The purpose of this measure is twofold: (1) to predict the extra time needed to test the software to achieve a specified objective; (2) to predict the expected reliability of the software when the testing is finished.

Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However, software reliability is different from hardware reliability in the sense that software does not wear out, burn out, or deteriorate, i.e., its reliability does not decrease with time. Moreover, software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease due to abrupt changes of its operational usage or incorrect modifications to the software. Software is also continuously modified throughout its life cycle. The malleability of software makes it inevitable for us to consider variable failure rates.

Unlike hardware faults which are mostly *physical faults,* software faults are *design faults,* which are harder to visualize, classify, detect,

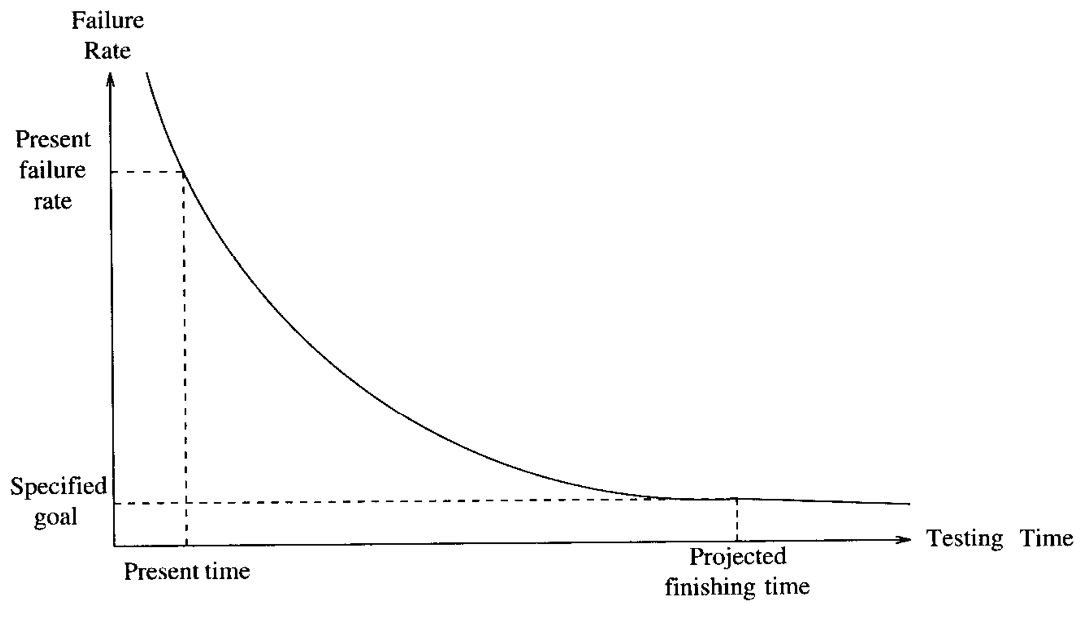

Figure 1.4 Basic ideas on software reliability modeling.

and correct. As a result, software reliability is a much more difficult measure to obtain and analyze than hardware reliability. Usually, hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of systems complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address nonstationary phenomena, such as reliability growth or reliability decrease, experienced in software. This makes software reliability a challenging problem that requires employing several methods of attack.

1.5 Technical Areas Related to the Book

Achieving highly reliable software from the customer's perspective is a demanding job to all software engineers and reliability engineers. Adopting a similar notation from [Lapr85, Aviz86] for system dependability, four technical methods are applicable for you to achieve reliable software systems:

1. *Fault prevention.* To avoid, *by construction,* fault occurrences.
2. *Fault removal.* To detect, *by verification and validation,* the existence of faults and eliminate them.
3. *Fault tolerance.* To provide, *by redundancy,* service complying with the specification in spite of faults having occurred or occurring.
4. *Fault/failure forecasting.* To estimate, *by evaluation,* the presence of faults and the occurrence and consequences of failures. This has been the main focus of software reliability modeling.

Detailed discussions regarding these technical areas are provided in the following sections. You can also refer to Chap. 2 (Sec. 2.2) for a complete list of dependability- and reliability-related concepts.

1.5.1 Fault prevention

The interactive refinement of the user's system requirement, the engineering of the software specification process, the use of good software design methods, the enforcement of a structured programming discipline, and the encouragement of writing clear code are the general approaches to prevent faults in the software. These guidelines have been, and will continue to be, the fundamental techniques in preventing software faults from being created.

Recently, *formal methods* have been attempted in the research community to attack the software quality problem. In formal-methods approaches, requirement specifications are developed and maintained using mathematically trackable languages and tools. Current studies in this area have been focused on language issues and environmental supports, which include at least the following goals: (1) executable specifications for systematic and precise evaluation, (2) proof mechanisms for software verification and validation, (3) development procedures that follow incremental refinement for step-by-step verification, and (4) every work item, be it a specification or a test case, is subject to mathematical verification for correctness and appropriateness.

Another fault-prevention technique, particularly popular in the software development community, is *software reuse*. The crucial measure of success in this area is the capability to prototype and evaluate reusable synthesis techniques. This is why *object-oriented paradigms* and techniques are receiving much attention nowadays—largely due to their inherent properties in enforcing software reuse.

1.5.2 Fault removal

When formal methods are in full swing, *formal design proofs* might be available to achieve mathematical proof of correctness for programs. Also, *fault-monitoring assertions* could be employed through executable specifications, and test cases could be automatically generated to achieve efficient software verification. However, before this happens, practitioners will have to rely mostly on *software testing* techniques to remove existing faults. Microsoft, for example, allocates as many software testers as software developers, and employs a buddy system which binds the developer of every software component to its tester for their daily work [Cusu95]. The key question to reliability engineers, then, is how to derive testing-quality measures (e.g., test-coverage factors) and establish their relationships to reliability.

Another practical fault removal scheme which has been widely implemented in industry is *formal inspection* [Faga76]. A formal inspection is a rigorous process focused on finding faults, correcting faults, and verifying the corrections. Formal inspection is carried out by a small group of peers with a vested interest in the work product during pretest phases of the life cycle. Many companies have acclaimed its success [Grad92].

1.5.3 Fault tolerance

Fault tolerance is the survival attribute of computing systems or software in their ability to deliver continuous service to their users in the presence of faults [Aviz78]. Software fault tolerance is concerned with all the techniques necessary to enable a system to tolerate software faults remaining in the system after its development. These software faults may or may not manifest themselves during system operations, but when they do, software fault tolerance techniques should provide the necessary mechanisms to the software system to prevent system failure from occurring.

In a single-version software environment, the techniques for partially tolerating software design faults include monitoring techniques, atomicity of actions, decision verification, and exception handling. In order to fully recover from activated design faults, multiple versions of software developed via *design diversity* [Aviz86] are introduced in which functionally equivalent yet independently developed software versions are applied in the system to provide ultimate tolerance to software design faults. The main approaches include the recovery blocks technique [Rand75], the N-version programming technique [Aviz77], and the N self-checking programming technique [Lapr87]. These approaches have found a wide range of applications in the aerospace industry, the nuclear power industry, the health care industry, the telecommunications industry, and the ground transportation industry.

1.5.4 Fault/failure forecasting

Fault/failure forecasting involves formulation of the fault/failure relationship, an understanding of the operational environment, the establishment of reliability models, the collection of failure data, the application of reliability models by tools, the selection of appropriate models, the analysis and interpretation of results, and the guidance for management decisions. The concepts and techniques laid out in [Musa87] have provided an excellent foundation for this area. Other reference texts include [Xie91, Neuf93]. Besides, the July 1992 issue of *IEEE Software,* the November 1993 issue of *IEEE Transactions on Software Engineering,* and the December 1994 issue of *IEEE Transactions on Reliability* are all devoted to this aspect of SRE. This handbook provides a comprehensive treatment of this subject.

1.5.5 Scope of this handbook

Due to the intrinsic complexity of modern software systems, software reliability engineers must apply a combination of the above methods for the delivery of reliable software systems. These four areas are also the main theme of the state of the art for software engineering, covering a wide range of disciplines. In addition to focusing on the fault/failure forecasting area, this book attempts to address the other three technical areas as well. However, instead of incorporating all possible techniques available in software engineering, this book examines and emphasizes mature as well as emerging techniques that could be *quantitatively* related to software reliability.

As a general guideline, most chapters of the book are concerned with fault/failure forecasting, in which Chaps. 1 to 5 provide technical foundations, while Chaps. 6, 7, 10, and 11 present project practices and experiences, and Chaps. 16 and 17 describe two emerging techniques. In addition, Chaps. 9 and 12 are related to fault prevention, and Chaps. 8 and 13 address fault removal techniques. (Fault prevention and removal techniques are the subject of discussion in many software engineering texts.) Finally, Chaps. 14 and 15 cover fault tolerance techniques and the associated modeling work. For a detailed treatment on software fault tolerance, interested readers are referred to [Lyu95].

The scope of the handbook is summarized in Table 1.3, which provides a guideline for using this book according to various subjects of interest, including the four technical areas we have discussed, and some special topics that you may want to study in depth. For example, if you are interested in the topic of software reliability modeling theory (topic 1), reading Chaps. 1, 2, 3, 4, 9, 10, 12, 14, and 16 is recommended. Note that topics 1 and 2 in Table 1.3 are mutually exclusive. So are topics 3 and 4, topics 5 and 6. Please note that the classification of the book chapters into various topics in Table 1.3 is for your reading convenience only. This classification is approximate and subjective.

1.6 Summary

The growing trend of software criticality and the unacceptable consequences of software failures force us to plead urgently for better software reliability engineering. This book codifies our knowledge of SRE and puts together a comprehensive and organized repository for our daily practice in software reliability. The structure of the book and key contents of each chapter are described. The definitions of major terms in SRE are provided, and fundamental concepts in software reliability modeling and measurement are discussed. Finally, the related technical areas in software engineering and some reading guidelines are provided for your convenience.

Problems

1.1 Some hardware faults are not physical faults and have a similar nature to software faults. What are they?

1.2 What are the main differences between software failures and hardware failures?

1.3 Give several examples of software faults and software failures.

1.4 Some people argue that the modeling technique for software reliability is similar to that of hardware reliability, while other people disagree. List the commonalities and differences between them.

TABLE 1.3 Reading Guideline for Various Technical Areas and Topics

	Technical Foundations					Practices and Experiences						Emerging Techniques					
Chapter	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Area 1									X			X					
Area 2								X					X				
Area 3														X	X		
Area 4	X	X	X	X	X	X	X			X	X					X	X
Topic 1	X	X	X	X					X	X		X		X		X	
Topic 2					X	X	X	X			X		X		X		X
Topic 3	X	X	X		X	X			X			X	X	X		X	
Topic 4				X			X	X		X	X				X		X
Topic 5	X	X			X	X			X				X	X	X	X	
Topic 6			X	X			X	X		X	X	X					X
Topic 7				X	X	X	X	X		X	X	X			X	X	X
Topic 8		X		X				X	X	X	X		X	X	X		X

NOTE:
Area 1—Fault prevention
Area 2—Fault removal
Area 3—Fault tolerance
Area 4—Fault/failure forecasting
Topic 1—Modeling theory
Topic 2—Modeling experience
Topic 3—Metrics
Topic 4—Measurement
Topic 5—Process issues
Topic 6—Product issues
Topic 7—Reliability data
Topic 8—Analysis techniques

1.5 Give a couple of examples for each of the definitions of failure severity levels. One is qualitative and one is quantitative.

1.6 What is the mapping relationship between faults and failures? Is it one-to-one mapping (one fault leading to one failure), one-to-many, many-to-one, or many-to-many? Discuss the mapping relationship in different conditions. What is the preferred mapping relationship? Why? How is it achieved?

1.7 The term *ultrareliability* has been used to denote highly reliable systems. This could be expressed, for example, as R (10 hour) = 0.9999999. That is, the probability that a system will fail in a 10-hour operation is 10^{-7}. Some people have proposed making this a reliability requirement for software. Discuss the implications of this kind of reliability requirement and its practicality.

1.8 What are the difficulties and issues involved in the data collection of failure-count data and time-between-failures data?

1.9 Regarding the failure data collection process, consider the following situations:

a. How do you adjust the failure times for an evolving program, i.e., a software program which changes over time through various releases?
b. How do you handle multiple sites or versions of the software?

1.10 Show that the time-between-failures data in Table 1.2 can be transformed to failure-count data in Table 1.1. Assuming random distribution, transform the failure-count data in Table 1.1 to time-between-failures data. Compare your results with Table 1.2.

1.11 For the data in Tables 1.1 and 1.2:

a. Calculate failure intensity at the end of each time period (for Table 1.1) or failure interval (for Table 1.2).
b. Plot the failure intensity quantities along with the time axis.
c. Try to fit a curve on the plots manually.
d. What are your estimates on (1) the failure rate of the next time period after observing the data in Table 1.1 and (2) the time to next failure after observing the data in Table 1.2?
e. What should be the relationship between the two estimates you obtained in *d?* Verify it.

1.12 Compare the MTTR measure for hardware and software and discuss the difference.

1.13 Refer to Example 1.2 and Fig. 1.2:

a. What is the failure rate of each component in Fig. 1.2? What is the reliability function of each component?
b. What assumption is made to calculate the MTTF for SYS2 in the triple-module redundant configuration? If the reliability function for SYS2 is $R_2(t)$, what is the reliability function for SYS2 in the triple-

module redundant configuration? How is its MTTF calculated? How is its MTTR calculated?

c. How is the overall system MTTF calculated? Verify that it is 125.9 hours when software failures are not considered, and that it is 11.9 minutes when software failures are considered.

d. How is the system MTTR calculated? Verify that it is 0.62 hours.

e. Does the triplication of SYS2 software help to improve its software MTTF? Why? If not, what techniques could be employed to improve the software MTTF?

1.14 *a.* What is the difference between reliability estimation and reliability prediction? Draw the application range of each technique in Fig. 1.4.

b. What is the difference between reliability prediction and early prediction? Summarize their differences in a comparison table.

1.15 Section 1.4 describes the concepts and constructions for software reliability models. It is important to identify which models are better than the others. Make a list of evaluation criteria for software reliability models.