SOFTWARE TESTING AND QUALITY ASSURANCE

Theory and Practice

KSHIRASAGAR NAIK

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo

PRIYADARSHI TRIPATHY

NEC Laboratories America, Inc.

A JOHN WILEY & SONS, INC., PUBLICATION

systems and operated. In many cases, during interoperability tests, users may require the hardware devices to be interchangeable, removable, or reconfigurable. Often, a system will have a set of commands or menus that allow users to make the configuration changes. The reconfiguration activities during interoperability tests are known as *configuration testing* [9]. Another kind of interoperability test is called a (*backward*) *compatibility test*. Compatibility tests verify that the system works the same way across different platforms, operating systems, and database management systems. Backward compatibility tests verify that the current software build flawlessly works with older version of platforms. As an example, let us consider a 1xEV-DO radio access network as shown in Figure 8.5. In this scenario, tests are designed to ensure the interoperability of the RNCs with the following products from different vendors: (i) PDSN, (ii) PDA with 1xEV-DO card, (iii) AAA server, (iv) PC with 1xEV-DO card, (v) laptop with 1xEV-DO card, (vi) routers from different vendors, (vii) BTS or RNC, and (viii) switches.

8.6 PERFORMANCE TESTS

Performance tests are designed to determine the performance of the actual system compared to the expected one. The performance metrics needed to be measured vary from application to application. An example of expected performance is: The response time should be less than 1 millisecond 90% of the time in an application of the “push-to-talk” type. Another example of expected performance is: A transaction in an on-line system requires a response of less than 1 second 90% of the time. One of the goals of router performance testing is to determine the system resource utilization, for maximum aggregation throughput rate considering zero drop packets. In this category, tests are designed to verify response time, execution time, throughput, resource utilization, and traffic rate.

For performance tests, one needs to be clear about the specific data to be captured in order to evaluate performance metrics. For example, if the objective is to evaluate the response time, then one needs to capture (i) end-to-end response time (as seen by external user), (ii) CPU time, (iii) network connection time, (iv) database access time, (v) network connection time, and (vi) waiting time.

Some examples of performance test objectives for an EMS server are as follows:

- Record the CPU and memory usage of the EMS server when 5, 10, 15, 20, and 25 traps per second are generated by the NEs. This test will validate the ability of the EMS server to receive and process those number of traps per second.
- Record the CPU and memory usage of the EMS server when log files of different sizes, say, 100, 150, 200, 250 and 300 kb, are transferred from NEs to the EMS server once every 15 minutes.

Some examples of performance test objectives of SNMP primitives are as follows:

- Calculate the response time of the Get primitive for a single varbind from a standard MIB or an enterprise MIB.

- Calculate the response time of the GetNext primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the GetBulk primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the Set primitive for a single varbind from a standard MIB or an enterprise MIB.

Some examples of performance test objectives of a 1xEV-DO Revision 0 are as follows:

- Measure the maximum BTS forward-link throughput.
- Measure the maximum BTS reverse-link throughput.
- Simultaneously generate maximum-rate BTS forward- and reverse-link data capacities.
- Generate the maximum number of permissible session setups per hour.
- Measure the AT-initiated connection setup delay.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 3-km/h mobility model.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 30-km/h mobility model.

The results of performance are evaluated for their acceptability. If the performance metric is unsatisfactory, then actions are taken to improve it. The performance improvement can be achieved by rewriting the code, allocating more resources, and redesigning the system.

8.7 SCALABILITY TESTS

All man-made artifacts have engineering limits. For example, a car can move at a certain maximum speed in the best of road conditions, a telephone switch can handle a certain maximum number of calls at any given moment, a router has a certain maximum number of interfaces, and so on. In this group, tests are designed to verify that the system can scale up to its engineering limits. A system may work in a limited-use scenario but may not scale up. The run time of a system may grow exponentially with demand and may eventually fail after a certain limit. The idea is to test the limit of the system, that is, the magnitude of demand that can be placed on the system while continuing to meet latency and throughput requirements. A system which works acceptably at one level of demand may not scale up to another level. Scaling tests are conducted to ensure that the system response time remains the same or increases by a small amount as the number of users are increased. Systems may scale until they reach one or more engineering limits. There are three major causes of these limitations:

i. Data storage limitations—limits on counter field size and allocated buffer space

ii. Network bandwidth limitations—Ethernet speed 10 Mbps and T1 card line rate 1.544 Mbps

iii. Speed limit—CPU speed in megahertz

Extrapolation is often used to predict the limit of scalability. The system is tested on an increasingly larger series of platforms or networks or with an increasingly larger series of workloads. Memory and CPU utilizations are measured and plotted against the size of the network or the size of the load. The trend is extrapolated from the measurable and known to the large-scale operation. As an example, for a database transaction system calculate the system performance, that is, CPU utilization and memory utilization for 100, 200, 400, and 800 transactions per second, then draw graphs of number of transactions against CPU and memory utilization. Extrapolate the measured results to 20,0000 transactions. The drawback in this technique is that the trend line may not be accurate. The system behavior may not degrade gradually and gracefully as the parameters are scaled up. Examples of scalability tests for a 1xEV-DO network are as follows:

- Verify that the EMS server can support the maximum number of NEs, say, 300, without any degradation in EMS performance.
- Verify that the maximum number of BTS, say, 200, can be homed onto one BSC.
- Verify that the maximum number of EV-DO sessions, say, 16,000, can be established on one RNC.
- Verify that the maximum number of EV-DO connections, say, 18,400, can be established on one RNC.
- Verify that the maximum BTS capacity for the three-sector configuration is 93 users per BTS.
- Verify the maximum softer handoff rate with acceptable number of call drops per BTS. Repeat the process every hour for 24 hours.
- Verify the maximum soft handoff rate with no call drops per BTS. Repeat the process for every hour for 24 hours.

8.8 STRESS TESTS

The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity. The system is deliberately stressed by pushing it to and beyond its specified limits. Stress tests include deliberate contention for scarce resources and testing for incompatibilities. It ensures that the system can perform acceptably under worst-case conditions under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked. Stress tests are targeted to bring out the problems associated with one or more of the following:

- Memory leak
- Buffer allocation and memory carving

One way to design a stress test is to impose the maximum limits on all system performance characteristics at the same time, such as the response time, availability, and throughput thresholds. This literally provides the set of worst-case conditions under which the system is still expected to operate acceptably.

The best way to identify system bottlenecks is to perform stress testing from different locations inside and outside the system. For example, individually test each component of the system, starting with the innermost components that go directly to the core of the system, progressively move outward, and finally test from remote locations far outside the system. Testing each link involves pushing it to its full-load capacity to determine the correct operation. After all the individual components are tested beyond their highest capacity, test the full system by simultaneously testing all links to the system at their highest capacity. The load can be deliberately and incrementally increased until the system eventually does fail; when the system fails, observe the causes and locations of failures. This information will be useful in designing later versions of the system; the usefulness lies in improving the robustness of the system or developing procedures for a disaster recovery plan. Some examples of stress tests of a 1xEV-DO network are as follows:

- Verify that repeated establishment and teardown of maximum telnet sessions to the BSC and BTS executed over 24 hours do not result in (i) leak in the number of buffers or amount of memory or (ii) significant increase in the degree of fragmentation of available memory. Tests should be done for both graceful and abrupt teardowns of the telnet session.
- Stress the two Ethernet interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Stress the four T1/E1 interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Verify that repeated establishment and teardown of AT connections through a BSC executed over 24 hours do not result in (i) leaks in the number of buffers or amount of memory and (ii) significant increase in the degree of fragmentation of available memory. The sessions remain established for the duration of the test.
- Verify that repeated soft and softer handoffs executed over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.
- Verify that repeated execution of all CLI commands over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.

Examples of stress tests of an SNMP agent are as follows:

- Verify that repeated walking of the MIBs via an SNMP executed over 24 hours do not result in leaks in number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory

- Verify that an SNMP agent can successfully respond to a GetBulk request that generates a large PDU, preferably of the maximum size, which is 8 kbytes under the following CPU utilization: 0, 50, and 90%.
- Verify that an SNMP agent can simultaneously handle multiple GetNext and GetBulk requests over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Get requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Set requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.

8.9 LOAD AND STABILITY TESTS

Load and stability tests are designed to ensure that the system remains stable for a long period of time under full load. A system might function flawlessly when tested by a few careful testers who exercise it in the intended manner. However, when a large number of users are introduced with incompatible systems and applications that run for months without restarting, a number of problems are likely to occur: (i) the system slows down, (ii) the system encounters functionality problems, (iii) the system silently fails over, and (iv) the system crashes altogether. Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load. This kind of testing helps one to understand the ways the system will fare in real-life situations. With such an understanding, one can anticipate and even prevent load-related problems. Often, operational profiles are used to guide load and stability testing [10]. The idea is to test the system the way it will be actually used in the field. The concept of *operation profile* is discussed in Chapter 15 on software reliability.

Examples of load and stability test objectives for an EMS server are as follows:

- Verify the EMS server performance during quick polling of the maximum number of nodes, say, 300. Document how long it takes to quick poll the 300 nodes. Monitor the CPU utilization during quick polling and verify that the results are within the acceptable range. The reader is reminded that quick polling is used to check whether or not a node is reachable by doing a ping on the node using the SNMP Get operation.
- Verify the EMS performance during full polling of the maximum number of nodes, say, 300. Document how long it takes to full poll the 300 nodes. Monitor the CPU utilization during full polling and verify that the results are within the acceptable range. Full polling is used to check the status and any configuration changes of the nodes that are managed by the server.

- Verify the EMS server behavior during an SNMP trap storm. Generate four traps per second from each of the 300 nodes. Monitor the CPU utilization during trap handling and verify that the results are within an acceptable range.
- Verify the EMS server's ability to perform software downloads to the maximum number of nodes, say, 300. Monitor CPU utilization during software download and verify that the results are within an acceptable range.
- Verify the EMS server's performance during log file transfers of the maximum number of nodes. Monitor the CPU utilization during log transfer and verify that the results are within an acceptable range.

In load and stability testing, the objective is to ensure that the system can operate on a large scale for several months, whereas, in stress testing, the objective is to break the system by overloading it to observe the locations and causes of failures.

8.10 RELIABILITY TESTS

Reliability tests are designed to measure the ability of the system to remain operational for long periods of time. The reliability of a system is typically expressed in terms of mean time to failure (MTTF). As we test the software and move through the system testing phase, we observe failures and try to remove the defects and continue testing. As this progresses, we record the time durations between successive failures. Let these successive time intervals be denoted by t_1, $t_2, \ldots, t_i$. The average of all the i time intervals is called the MTTF. After a failure is observed, the developers analyze and fix the defects, which consumes some time—let us call this interval the *repair* time. The average of all the repair times is known as the mean time to repair (MTTR). Now we can calculate a value called mean time between failures (MTBF) as $\text{MTBF} = \text{MTTF} + \text{MTTR}$. The random testing technique discussed in Chapter 9 is used for reliability measurement. Software reliability modeling and testing are discussed in Chapter 15 in detail.

8.11 REGRESSION TESTS

In this category, new tests are not designed. Instead, test cases are selected from the existing pool and executed to ensure that nothing is broken in the new version of the software. The main idea in regression testing is to verify that no defect has been introduced into the unchanged portion of a system due to changes made elsewhere in the system. During system testing, many defects are revealed and the code is modified to fix those defects. As a result of modifying the code, one of four different scenarios can occur for each fix [11]:

- The reported defect is fixed.
- The reported defect could not be fixed in spite of making an effort.

CHAPTER 15

Software Reliability

Failure is only postponed success as long as courage "coaches" ambition. The habit of persistence is the habit of victory.
— *Herbert Kaufman*

15.1 WHAT IS RELIABILITY?

The concept of *reliability* is very broad, and it can be applied whenever someone expects something or someone else to "behave" in a certain way. For example, a lighting switch is expected to stay in one of two states—on and off—as set by its user. If a lighting switch causes a lamp to flicker even when the power supply is stable and the connecting cables are fault free, we say that the switch has turned unreliable. As a customer, we expect a new switch not to cause any flicker. We say that the switch is faulty if it causes a lamp to flicker. Moreover, a new switch may operate in a fault-free manner for several years before it starts causing the lamp to flicker. In that case we say that wear and tear have caused the switch to malfunction, and thus the switch has become unreliable. Therefore, the concept of fault is intertwined with the concept of reliability.

An initially fault-free hardware system behaves in a reliable manner simply because it operates as expected due to absence of faults in its design and manufacturing. Once the new hardware system starts functioning, wear and tear can cause it to develop faults, thereby causing the system to behave in an unexpected, or unreliable, manner. The sources of failures in software systems are design and implementation faults. Software failures usually occur when a program is executed in an environment that it was not developed for or tested for.

Reliability is one of the metrics used to measure the quality of a software system. It is arguably the most important quality factor sought in a product. Reliability is a user-oriented quality factor relating to system *operation*, and it takes into account the frequency of system failure. Intuitively, if the users of a system rarely experience system failure, then the system is considered to be more reliable than one that fails more often. The more and more failures of a system are observed by users, the less and less the system is perceived to be reliable. Ideally, if there are no faults in a software system, users will never experience system failure, thereby considering the system to be highly reliable. Constructing a "correct" system, that is, a fault-free system, is a difficult task by itself given that real-life systems are inherently complex. The problem of constructing a fault-free system becomes more difficult when real-life factors are considered. For example, developing software systems is largely an economic activity, and companies may rarely have all the time and money needed to produce a highly reliable system even when they have a team of highly qualified and experienced personnel in the best case. The concept of *market window* may not allow a company to make an effort at producing a "correct" system. A market window is considered to be a time interval available for the introduction of a product before the product is surpassed in capability or cost by another from a competing vendor. Economic considerations may drive companies to trade off reliability in favor of cutting cost and meeting delivery schedule.

Given that there are an unknown number of faults in a delivered system and users can tolerate some failures, system reliability is best represented as a continuous variable rather than a Boolean variable. More efforts put into the development phase generally lead to a higher degree of reliability. Conversely, less effort produces systems with lower reliability. Thus, the concept of reliability can be used in examining the significance of trends, in setting goals, and in predicting when those goals will be achieved. For example, developers may be interested in knowing how a certain development process, how the length of system testing, or how a design review technique impacts software reliability. Developers may be interested in knowing the rate of system failure in a certain operational environment so as to be able to decide when to release the product.

Software maintenance involves making different kinds of changes to the system, namely, changes to requirements, changes to the design, changes to source code, and changes to test cases. While making those changes, the software goes through a phase of instability. The instability is in the form of reduced reliability of the system. The reliability level of a system decreases during system maintenance because additional defects may be introduced while making all those kinds of changes to the system. Intuitively, the smaller the amount of changes made to a system, the lesser the degradation in the current reliability level of the system. On the other hand, too many changes being simultaneously made to the system may significantly degrade the reliability level. Thus, the amount of changes that should be made to a product at a given time is dependent upon how much reliability one is ready to sacrifice for the moment.

15.1.1 Fault and Failure

The notions of *fault, failure*, and *time* are central to our understanding of reliability. In general, if a user never observes failures, the system is considered to be very reliable. On the other hand, a frequently failing system is considered to be highly unreliable. Therefore, we revisit the terms fault and failure in this section. A failure is said to occur if the *observable* outcome of a program execution is different from the expected outcome. The concept of observable outcome is very broad, and it encompasses a variety of things, such as values produced, values communicated, performance demonstrated, and so on. The expected outcomes of program executions are specified as system requirements. System requirements are generally stated in explicit form in a requirements document. Sometimes, due to the complex nature of software systems, it is extremely difficult to explicitly state all the desired requirements. Where it is not possible to state all the requirements in an explicit manner, we still expect a system to behave in certain ways. Thus, while considering what is an expected outcome of program execution, one must take into account both the explicitly stated and the implicitly expected system requirements. Two characteristics of failures are as follows: (i) failures are associated with actual program executions and (ii) failures are observable concepts.

The adjudged cause of a failure is called a fault. While constructing a software system, a designer may mistakenly introduce a fault into the system. A fault can be introduced by having a defective block of code, a missing block of code for an unforeseen execution scenario, and so on. The mere presence of faults in a system does not cause failure. Rather, one or more defective portions of a system must execute for a fault to manifest itself as a system failure. One fault can cause more than one failure depending upon how the system executes the faulty code [1].

15.1.2 Time

Time plays a key role in modeling software reliability metrics. Let us go back to the concept of reliability of a lighting switch. Assume that a switch causes a lamp to flicker for a couple seconds, and let the average time gap between two successive flickers be six months. Here, flickering of the lamp can be considered as an observable failure of the switch. Though users may be irritated by such flickers, some may still consider the switch to be reliable because of the long time gap between two flickers. A time gap of six months can be considered to be long because a user may not remember when the lamp flickered last. However, if the lamp flickers a few times everyday, the switch appears less reliable and becomes a candidate for replacement. Thus, the notion of time is important in understanding the concept of reliability.

There are two commonly considered time models in the study of software reliability:

- Execution time (τ)
- Calendar time (t)

The *execution time* for a software system is the actual time spent by a processor in executing the instructions of the software system. Sometimes a processor may execute code from different software systems, and therefore, their individual execution times must be considered separately. On the one hand, there are many software systems which are expected to run continuously for months and years without developing failure. Examples of such systems are telephone switching software, air-traffic control software, and software for monitoring power plants. On the other hand, there are software systems which run for a while and terminate upon completion of their tasks. For example, a compiler terminates after compiling a program. Though the concept of software reliability can be applied to all kinds of software systems, the ones running continuously draw more attention.

Calendar time is the time more commonly understood and followed in everyday life by all, including software engineers and project managers. Since observation of failures, fixing the corresponding faults, and replacing an old version of a software with a new one are intertwined in the life cycle of software systems, calendar time is useful for pragmatic reasons such as engineers leaving a company, engineers going on leave, and so on.

A third time model occasionally referred to is the *clock time* of a software system. Clock time refers to the elapsed time between the start and the end of program execution. Clock time includes the wait time of the software system and execution times of other software systems. Clock time does not include system shutdown.

In order to have a better understanding of the reliability of a software, one can ask the following questions in terms of failure and time:

- What is the time interval between two successive failures?
- How many failures have been observed within a certain time interval, for example, in the past one month or one year?
- What is the total number of failures observed so far?

The answers to the above questions give an indication of the quality level of a software. One can ask the above questions at any instant after code is developed. By asking the above questions during development, a test manager can monitor how the quality of the software is improving. On the other hand, by answering the questions during the operation of the system, one can know the delivered quality of the product.

15.1.3 Time Interval between Failures

One can infer several characteristics of a software system by monitoring the time interval between successive failures as follows:

- A small time interval between successive failures tells us that the software system is failing frequently, and hence the reliability level is too low. This can happen during system testing or even while the system is in operation. If the time interval between successive failures is long, the reliability is perceived to be high, in spite of the occasional system failure. The concepts of "short" and "long" time intervals between successive failures is a matter of user perception that is largely defined by the consequences of the failures. For example, if the operating system of a personal computer crashes about once in a year, the users can still consider the system to be reliable. On the other hand, if an air-traffic control software installed in a major international airport crashes once in a year, the system will be considered to be highly unreliable.

  In the hardware industry, a number of reliability metrics have been identified by considering the instants when failures occur and the instants when the corresponding faults are repaired. The three commonly used metrics based on time intervals are the mean time to failure (MTTF), the mean time to repair (MTTR), and the mean time between failure (MTBF). When a failure occurs, it takes a certain amount of time to repair the system. The mean of all the repair times is the MTTR. We assume that a system is not in operation while it is being repaired. Thus, the mean of all the time intervals between the completion of a repair task and the occurrence of the next failure is the MTTF. The mean of the time intervals between successive failures is the (MTBF). The terms MTTR, MTTF, and MTBF are illustrated in Figure 15.1. A useful relationship between the three metrics can be stated as $\text{MTBF} = \text{MTTF} + \text{MTTR}$. It is easy to verify the relationship by considering the time intervals in Figure 15.1.

- At the beginning of system-level testing, usually a large number of failures are observed with small time intervals between successive failures. As system testing continues and faults are actually fixed, the time interval between successive failures increases, thereby giving an evidence that the reliability of the product is increasing. By monitoring the time interval between successive failures, one gets an idea about the reliability of the software system.

15.1.4 Counting Failures in Periodic Intervals

Useful information can be gathered by monitoring the cumulative failure count on a periodic basis. For example, we record the cumulative failure count every

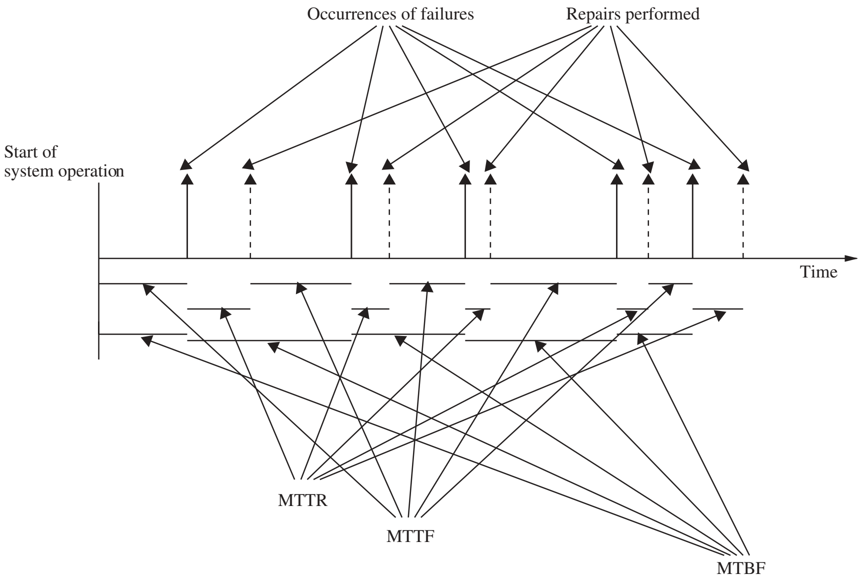

Figure 15.1 Relationship between MTTR, MTTF, and MTBF.

month, plot it as a bar chart, and observe the pattern. Such monitoring can be done during system testing as well as while a system is in operation. During system testing, monitoring can be done at a fast pace, such as once per week, because of the large number of failures observed during testing. Development engineers and managers would like to observe the improvement in the reliability of the system under development. For a system already in operation, monitoring can be done once or twice per year. One can observe a number of system failures during its operation because of two reasons. First, there is a residual number of faults in most systems and systems might not have been tested in their real execution environments. Second, additional faults might be inadvertently introduced into a system during system maintenance.

15.1.5 Failure Intensity No es necesario

By counting the total number of failures observed so far and plotting the data as a function of time, one can observe the change in the reliability of the system. This information is useful while a software is undergoing system-level testing as well as while a system is in operation. Ideally, the reliability level should reach a stable value after a while. A rising graph of the cumulative number of failures tells us that there are more faults in the system, and hence the system is perceived to be unreliable. The rate of rising of the graph is the rate at which failures are being observed. If the rate of rising is very small, we know that failures are infrequently being observed.

Two failure-related quantities that are generally used in defining software reliability are *cumulative failure*, denoted by the symbol μ, and *failure intensity*, denoted by the symbol λ. Failure intensity is expressed as the number of failures observed per unit of time. One can consider execution time τ or calendar time t. In this chapter, we consider the execution time τ in modeling software reliability. However, one can use calendar time as well. For example, if 10 failures are observed per hour of CPU time, then we express failure intensity as 10 failures per CPU hour. Similarly, if 30 failures are observed every two hours, then we express failure intensity as 15 (=30/2) failures per CPU hour.

Both μ and λ are functions of time. The cumulative failure count μ is a function of time because as time passes the total failure count monotonically increases. Moreover, λ is a function of time because as more and more failures are observed and the corresponding faults are actually fixed, we observe fewer and fewer number of failures per unit of execution time. Ideally, λ decreases with τ. But, in reality, λ can be a random function of τ as developers may introduce more faults while attempting to fix a known fault. The quantities $\mu(\tau)$ and $\lambda(\tau)$ are explained as follows:

- The quantity $\mu(\tau)$ denotes the total number of failures observed until execution time τ from the beginning of system execution.
- The quantity $\lambda(\tau)$ denotes the number of failures observed per unit time after τ time units of executing the system from the beginning. This quantity is also called the failure intensity observed at time τ.

Given these definitions, it is not difficult to establish the following relationship between $\mu(\tau)$ and $\lambda(\tau)$:

$$\lambda(\tau) = \frac{d\mu(\tau)}{d\tau}$$

Conversely, if we know the *failure intensity function*, we obtain the *mean value function* [2]:

$$\mu(\tau) = \int_0^\tau \lambda(x)\, dx$$

15.2 DEFINITIONS OF SOFTWARE RELIABILITY

There are two commonly used software reliability metrics, namely, the *probability of failure-free operation* and the *failure intensity*. The concept of failure is common to both definitions. One metric is probabilistic in nature, whereas the other is an absolute one. The two definitions are not contradictory. Rather, both can be simultaneously applied to the same software system without any inconsistency between them. User expectations from different systems may be different, and therefore it is more useful to apply one of the two to a given system. In the following, first the two metrics are defined, followed by examples of their applications to different systems.

15.2.1 First Definition of Software Reliability

Definition. Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.

The key elements of the above definition of reliability are as follows:

- Probability of failure-free operation
- Length of time of failure-free operation
- A given execution environment

Software reliability is expressed as a continuous random variable due to the fact that most large software systems do have some unknown number of faults, and they can fail anytime depending upon their execution pattern. Therefore, it is useful to represent software reliability as a probabilistic quantity. Though software systems do fail from time to time, the users are interested in completing their tasks most of the time. The need for failure-free operation for a certain length of time is important due to the fact that a user intends to complete a task that demands some length of execution time. For example, consider an inventory management software system used by a large store. Assume that the store opens from 8 AM to 8 PM, and they shut down the system after store hours. The store owner expects the inventory system to be up and running for at least 12 hours without developing any failure.

One can specify the reliability of a software system running on a personal computer (PC) as follows. Assume that an office secretary turns on his or her PC every morning at 8.30 AM and turns it off at 4:30 PM before leaving for home. The secretary expects that the PC will run for eight hours without developing any failure. If he or she comes to the office for 200 days in a year and observes that the PC crashes five times on different days in a year for a few years, we can say that the probability of failure-free operation of the PC for eight hours was 0.975 [=(200 − 5)/200].

The third element in the definition of software reliability is an *execution environment*. An execution environment refers to how a user operates a software system. Not all users operate a software system in the same way. As an example, let us consider the case of a word processor. One group of users may be processing documents of small size, say 50 pages. A second group of users may be processing documents of large size, say, 1000 pages. Clearly, the two groups of users offer two different execution environments to the word processor. For example, to process documents of large size, the software is likely to invoke a part of its code for managing memory space which may not be needed to process a document of small size. The idea of how users operate a system gives rise to the concept of execution environment.

The concept of execution environment is an essential part of the definition of software reliability. Consider that a software system supports 10 different functions $f_1, \ldots, f_{10}$ and there are two groups of users. One group of users use only functions $f_1, \ldots, f_7$, and the second group uses all the functions. Let functions $f_1, \ldots, f_7$ be fault free, but there are faults in functions f_8, f_9, and f_{10}. Consequently, the first group of users will never observe any failure simply because their operations of the software system do not involve the faulty components of the software. From the viewpoint of the first group of users, the probability of failure-free operation of the software is 1.0. On the other hand, the second group of users will observe failure from time to time depending upon how frequently they use functions f_8, f_9, and f_{10}. Consequently, the level of reliability perceived by the second group of users is lower than that perceived by the first group of users. Therefore, the concept of execution environment is crucial to the definition of software reliability. In a later part of this chapter, the concept of operational profile will be discussed to describe execution environment.

15.2.2 Second Definition of Software Reliability

No es necesario

Definition. Failure intensity is a measure of the reliability of a software system operating in a given environment.

According to the second definition, the lower the failure intensity of a software system, the higher is its reliability. To represent the current reliability level of a software system, one simply states the failure intensity of the system. For example, let a software system be in its system testing phase where test engineers are observing failures at the rate of 2 failures per eight hours of system execution.

Then, one can state the current level of the reliability of the system as 0.25 failure per hour. The figure 0.25 failure per hour is obtained by dividing 2 failures by eight hours.

No es necesario

15.2.3 Comparing the Definitions of Software Reliability

The first definition of software reliability emphasizes the importance of a software system operating without failure for a certain minimum length of time to complete a transaction. Here, reliability is a measure of what fraction of the total number of transactions a system is able to complete successfully. Let us assume that an autonomous robot is sent out to explore the underwater sea bed, and it takes three days to complete each round of exploration. In this case, we expect that the on-board software system resident in the robot is expected to run for at least three days continuously so that a round of exploration is successfully completed. If a robot system has successfully completed 99% of all the exploration rounds, we say that the software system's reliability is 0.99.

The second definition of reliability simply requires that there be as few failures as possible. Such an expectation takes the view that the risk of failure at any instant of time is of significant importance. This is very much the case in the operation of an air-traffic control system in an airport. This case is also applicable to the operation of telephone switches. In this definition, it does not matter for how long the system has been operating without failure. Rather, the very occurrence of a failure is of much significance. For example, the failing of a traffic control system in an airport can lead to a major catastrophe.

15.3 FACTORS INFLUENCING SOFTWARE RELIABILITY

A user's perception of the reliability of a software system depends upon two categories of information, namely, (i) the number of faults present in the software and (ii) the ways users operate the system. The second category of information is known as the *operational profile* of the system. The number of faults introduced in a system and the developers' inability to detect many of those faults depend upon several factors as explained below:

- **Size and Complexity of Code:** The number of LOC in a software system is a measure of its size. Large software systems with hundreds of thousands of LOC tend to have more faults than smaller systems. The likelihood of faults in a large system, because of more module interfaces, is higher. The more number of conditional statements the code contains, the more complex the system is considered to be. Due to the economic considerations in software development, one may not have much time to completely understand a large system. Consequently, faults are introduced into the system in all its development phases. Similarly, while removing faults from a large system, new faults may be introduced.

- **Characteristics of Development Process:** Much progress has been made in the past few decades in the field of software engineering. New techniques and tools have been developed to capture system requirements, to design software systems, to implement designs, and to test systems. For example, formal methods, such as SDL (Specification and Description Language) and UML (Unified Modeling Language), are used to specify the requirements of complex, real-time systems. Code review techniques have been developed to detect design and implementation faults. Test tools are available to assist programmers in their unit-level and system-level testing. By developing a system under a larger quality control umbrella in the form of embracing the above software engineering techniques and tools, the number of remaining faults in software systems can be reduced.
- **Education, Experience, and Training of Personnel:** The information technology industry has seen tremendous growth in the past 20 years or so. However, education of software engineering as a separate discipline of study at the undergraduate level is only emerging now on a large scale. It is not unusual to find many personnel with little training to be writing, modifying, and testing code in large projects. Lack of desired skills in personnel can cause a system to have a larger number of faults.
- **Operational Environment:** Detection of faults remaining in a software system depends upon a test engineer's ability to execute the system in its actual operational environment. If a test engineer fails to operate a system in the same manner the users will do, it is very likely that faults will go undetected. Therefore, test engineers must understand the ways the users will operate a system. Because of a lack of sufficient time and lack of experience on the part of development and test engineers, faults can be introduced into a system, and those faults can go undetected during testing.

Therefore, software reliability is determined by a complex combination of a number of factors with wide-ranging characteristics. Ideally, from the viewpoint of modeling, it is useful to have a mathematical model that takes in the influencing parameters and gives us a concrete value of the level of reliability of a software system. However, such an ideal model has not been devised because of the sheer complex nature of the problem. In reality, the reliability models studied in the literature consider a small number of influencing parameters. Despite the limited scope of those models, we gain much insight into the complex concept of software reliability. For example, the more the test engineers execute a system during system-level testing, the more is the likelihood of observing failures. By identifying and fixing the faults causing those failures, development engineers can improve the reliability level of the system. Thus, the amount of testing time directly influences the level of software reliability. Consequently, a reliability model can be developed where *system testing time*, or simply *time*, is the independent variable and level of reliability is the dependent variable in the reliability model. By using such a model, one can predict the length of system testing time required to achieve a given level of software reliability [3].

15.4 APPLICATIONS OF SOFTWARE RELIABILITY

Reliability is a quantitative measure of the failure-related quality aspect of a software system. A number of factors, as explained in Section 15.3, influence the reliability level of software systems. It is natural to evaluate the effectiveness of the influencing factors.

15.4.1 Comparison of Software Engineering Technologies

To develop higher quality software systems in a cost-effective manner, a number of technologies have been introduced. For example, there are numerous software development processes, such as the *waterfall* model [4], the *spiral* model [5], the *prototyping* model [6], the *eXtreme Programming* model [7], and the *Scrum* model [8, 9]. In terms of introduction of testing tools, many techniques have been studied to generate test cases. Several techniques have been introduced to capture customer requirements, such as entity relationship diagrams, data flow diagrams, and the UML [10, 11]. When a new technology is developed, it is necessary to evaluate its effectiveness before fully embracing it. Three useful criteria for evaluating a technology from the management viewpoint are as follows:

- What is the cost of adopting the technology?
- How does the new technology affect development schedule?
- What is the return from the new technology in terms of software quality?

The concept of software reliability can be used to evaluate a new technology in terms of its usefulness in allowing software developers to produce higher quality software. For example, consider two technologies M_1 and M_2. If an application is developed using technology M_1 to generate a system S_1, and technology M_2 is used to generate another system S_2 to implement the same application, it is useful to observe the difference in the reliability levels of S_1 and S_2. By monitoring the reliability levels of the two systems S_1 and S_2 for the same application, managers can observe what technology is more effective in producing software systems of higher reliability.

15.4.2 Measuring the Progress of System Testing

Measuring the progress of software development is central to managing a project. An important question in project management is: *How much has been completed?* An answer to the question lets us know if progress is being made as planned. It is important to monitor the progress of system testing, because it consumes a significant portion of money and time. Two useful metrics to monitor the progress of system-level testing are as follows:

- Percentage of test cases executed,
- Percentage of successful execution of high-priority functional tests

The concept of software reliability can be used to measure the progress of system testing. For example, one can monitor the failure intensity of the SUT to know where on the reliability scale the system lies at this moment. If the current failure intensity is much larger than a tolerable failure intensity at the time of its release, we know that much work remains to be done. On the other hand, if the difference between the current failure intensity and the desired failure intensity is very small, we realize that we are close to reaching the goal. Thus, the concept of software reliability can be used in an objective manner to measure how much progress has been made in system-level testing [12, 13].

15.4.3 Controlling the System in Operation

The reliability of a system usually reduces as a result of maintenance works. The larger the amount of change made to a system, the larger is the reduction in the reliability level of the system. For example, assume that a system has k number of faults per 1000 lines of code at the beginning of system testing. The value of k can be obtained from statistical measures. By adding N lines of code to the system as a part of a maintenance activity, statistically, $Nk/1000$ faults are introduced into the system. Introduction of the new faults will decrease the reliability of the system, and it will require prolonged system testing to detect the faults and raise system reliability by fixing those faults. A project manager may put a limit on the amount of change in system reliability for each kind of maintenance activity. Therefore, the size of a maintenance work can be determined by the amount of system reliability that can be sacrificed for a while.

15.4.4 Better Insight into Software Development Process

The concept of reliability allows us to quantify the failure-related quality aspect of a software system. Quantification of the quality aspect of software systems gives developers and managers a better insight into the process of software development. For example, by observing the failure intensity at the beginning of system testing, a test manager may be able to make an informed decision regarding how long system testing may take to bring down the failure intensity to an acceptable level. In other words, managers will be capable of making informed decisions.

15.5 OPERATIONAL PROFILES

The notion of *operational profiles*, or *usage profiles*, was developed at AT&T Bell Laboratories [14] and IBM [15, 16] independently. As the name suggests, an operational profile describes how actual users operate a system. An operational profile is a quantitative characterization of how a system will be used. However, for accurate estimation of the reliability of a system one must test it by considering how it will actually be used in the field.

15.5.1 Operation

An *operation* is a major system-level logical task of short duration which returns control to the initiator when the operation is complete and whose processing is substantially different from other operations. In the following, the key characteristics of an operation are explained:

- *Major* means that an operation is related to a functional requirement or feature of a software system.
- An operation is a *logical* concept in the sense that it involves software, hardware, and user actions. Different actions may exist as different segments of processing. The different processing time segments can be contiguous, noncontiguous, sequential, or concurrent.
- *Short duration* means that a software system is handling hundreds of operations per hour.
- *Substantially different processing* means that an operation is considered to be an entity in the form of some lines of source code, and there is a high probability that such an entity contains a fault not found in another entity.

15.5.2 Representation of Operational Profile

The operational profile of a system is the set of operations supported by the system and their probabilities of occurrence. For example, there are three operations, namely, A, B, and C, supported by a system, and they occur 50, 30, and 2% of the time. Then the operational profile is represented by the set $\{(A, 0.5), (B, 0.3), (C, 0.2)\}$. Operational profiles are represented in two alternative ways:

- *Tabular* representation
- *Graphical* representation

Tabular Representation of Operational Profile An operational profile can be represented in a tabular form with three columns as shown in Table 15.1. The example is about a library information system. The first column lists the names of the operations, the second column gives the frequency of using the operations, and the final column shows the probability of using the operations. We list two kinds of book returned operations: books returned in time and books returned late. The two operations involve some separate processing because a book returned late may incur some penalty. The book renewed operation is a combination of one of the two book returned operations and a book checked out operation.

Graphical Representation of Operational Profile An operational profile can also be represented in graphical form as shown in Figure 15.2. In a graphical form, an operational profile is represented as a tree structure consisting of nodes and branches. Nodes represent attributes of operations, and branches represent values of attributes with the associated probability of occurrence. In Figure 15.2, four

TABLE 15.1 Example of Operational Profile of Library Information System

Operation	Operations per Hour	Probability
Book checked out	450	0.45
Book returned in time	324	0.324
Book renewed	81	0.081
Book returned late	36	0.036
Book reported lost	9	0.009
⋮	⋮	⋮
Total	1000	1.0

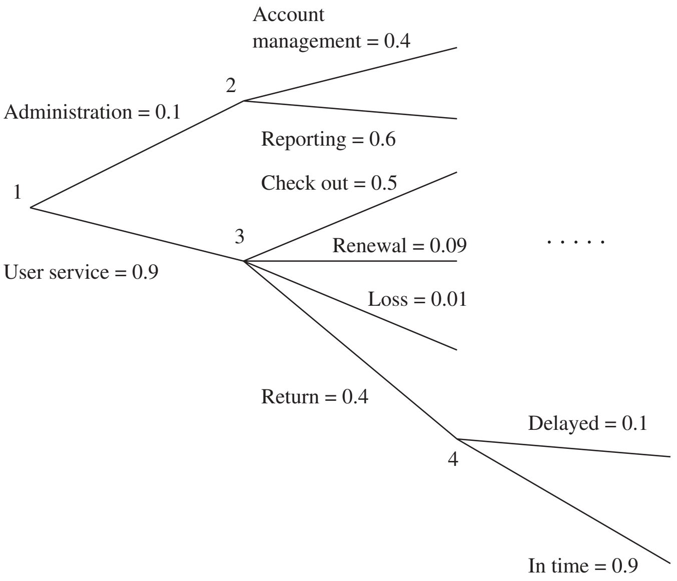

Figure 15.2 Graphical representation of operational profile of library information system.

nodes, namely 1, 2, 3, and 4 are shown. Node 1 represents the scope attribute of operations with two values, administration and user service. The scope of an operation refers to whether an operation is for the administration of the information system or for providing services to users. The probability of occurrence of the administration value of the scope attribute of an operation is 0.1, and the probability of occurrence of the user service value is 0.9. Node 2 represents an administration operation, whereas node 3 represents a user operation. There are two attributes of the administration operation, namely, account management and reporting, and the associated probabilities of occurrence are 0.4 and 0.6, respectively. There are four attributes of the user operations, namely, check out, renewal, loss, and return, with their associated probabilities of occurrence being 0.5, 0.09, 0.01, and 0.4, respectively. Node 4 represents a return operation with two attribute values, namely delayed and in-time, with their probabilities of occurrence being 0.1 and 0.9, respectively.

It is useful to note that a tabular form of an operational profile can be easily generated from a graphical form by considering all possible paths in the graphical form and multiplying the probabilities appearing on each path. For example, the probability of the book returned late operation (0.036) in Table 15.1 can be obtained from the graphical form shown in Figure 15.2 by multiplying the probabilities associated with user service (0.9), return (0.4), and delayed (0.1).

Choosing Tabular Form or Graphical Form The tabular form of an operational profile can be easily obtained from the graphical form. It is easy to specify the operational profile in a tabular form if a system involves only a small number of operations. However, the graphical form is preferred if a system involves a large number of operations. Operations that can be easily described as sequences of smaller processing steps are better suited for the graphical form. Moreover, it is easy to identify missing operations in a graphical form.

Using Operational Profile During System Testing The notion of operational profiles was created to guide test engineers in selecting test cases. Ideally, at least one test case should be selected to test an operation, and all the operations should be covered during testing. Since software reliability is very much tied with the concept of failure intensity, a software system with better reliability can be produced within a given amount of time by testing the more frequently used operations first. In reality, projects overrun their schedules, and there may be a tendency to deliver products without adequate testing. An operational profile can be used in making a decision concerning how much to test and what portions of a software system should receive more attention. The ways test engineers select test cases to operate a system may significantly differ from the ways actual users operate a system. However, for accurate estimation of the reliability of a system, test the system in the same way it will actually be used in the field.

Other Uses of Operational Profiles The operational profile of a system can be used in a number of ways throughout the life-cycle model of a software system as follows:

- As a guiding document in designing user interface. The more frequently used operations should be easy to learn and easy to use.
- In developing a version of a system for early release. The early-release version can contain the more frequently used operations.
- To determine where to put more resources for system development. For example, more resources can be allocated to the development of those operations which are used more frequently.
- As a guiding document for organizing user manuals. For example, the more frequently used operations are described earlier than the rest.

15.6 RELIABILITY MODELS

No es necesario

The reliability models are developed based on the following assumptions:

- Faults in the program are independent.
- Execution time between failures is large with respect to instruction execution time.
- Potential *test space* covers its *use space*.
- The set of inputs per test run is randomly selected.
- All failures are observed.
- The fault causing a failure is immediately fixed or else its reoccurrence is not counted again.

To understand the idea of fault independence in the first assumption, we can consider a mapping between the set of faults in a system and the set of failures caused by the faults. Faults are said to be independent if there is a one-to-one or one-to-many relationship between faults and failures. Thus, if faults are independent and a fault is fixed, then the corresponding single or multiple failures are no longer observed. In a many-to-one relationship, fixing one fault will not eliminate the corresponding failure; all the faults need to be fixed to eliminate the failure.

The second assumption concerning the length of execution time between failures tells us that the system does not fail too often. A reasonably stable system is a prerequisite for the reliability models to be valid. No meaningful prediction can be done if a system fails too often.

The third assumption, concerning test space and use space, implies that a system be tested by keeping in mind how it will be used. For example, consider a telephone system comprising three features: (i) call processing, (ii) billing, and (iii) administrative operation. Assume that there are 15 basic operations to support each feature. Even though there are a total of 45 operations, actual operation of the system can produce a large number of execution scenarios. Call establishment between two phones is a basic operation in the call processing group, and updating a customer profile is a basic operation in the administrative group. Those two operations can be tested separately. However, there may be a need to update a customer's profile while the customer is in the middle of a call. Testing such an operational situation is different from individually testing the two basic operations. The usefulness of a reliability model depends on whether or not a system is tested by considering the use space of the system that can be characterized by an operational profile.

The fourth assumption emphasizes the need to select test input randomly. For example, referring to the telephone system in the explanation of the third assumption, to test the call establishment function, we randomly select the destination number. Randomness in the selection process reduces any bias in favor of certain groups of test data.

The fifth assumption concerning failures simply tells us to consider only the final discrepancies between actual system behavior and expected system behavior.

Even if a system is in an erroneous state, there may not be a system failure if the system has been designed to tolerate faults. Therefore, only the observed failures are taken into consideration, rather than the possibility of failures.

The sixth assumption tells us how to count failures. When a failure is observed, it is assumed that the corresponding fault is detected and fixed. Because of the first assumption that faults are independent, the same failure is not observed due to another, unknown fault. Thus, we count a failure once whether or not the corresponding fault is immediately fixed.

The details of the development of two mathematical models of reliability are presented. In these two models failure intensity as a measure of reliability is expressed as a function of execution time. That is, an expression for $\lambda(\tau)$ is developed. The models are developed based on the following intuitive idea: *As we observe another system failure and the corresponding fault is fixed, there will be a fewer number of faults remaining in the system and the failure intensity of the system will be smaller with each fault fixed. In other words, as the cumulative failure count increases, the failure intensity decreases*.

In the above intuitive idea, an important concept is the characterization of the *decrement* in failure intensity as a function of cumulative failure count. In this chapter, two reliability models are developed by considering two decrement processes as follows:

Decrement Process 1: The decrease in failure intensity after observing a failure and fixing the corresponding fault is constant. The reliability model developed using this model of failure intensity decrement is called the basic model.

Decrement Process 2: The decrease in failure intensity after observing a failure and fixing the corresponding fault is smaller than the previous decrease. In other words, fixing a fault leading to an earlier failure causes the failure intensity to be reduced by a larger amount than fixing a fault causing a later failure. Therefore, failure intensity is exponential in the number of failures observed. The reliability model developed using this model of failure intensity decrement is called the logarithmic model.

In the development of the two reliability models, the following notation is used:

- μ denotes the mean number of failures observed.
- λ denotes the mean failure intensity.
- λ_0 denotes the *initial* failure intensity observed at the beginning of system-level testing.
- ν_0 denotes the total number of system failures that we expect to observe over infinite time starting from the beginning of system-level testing.
- θ denotes the decrease in failure intensity in the logarithmic model. This term will be further explained in the discussion of logarithmic model below.

Basic Model The constant decrement in failure intensity per failure observed is illustrated in Figure 15.3, where the straight line represents the failure decrement

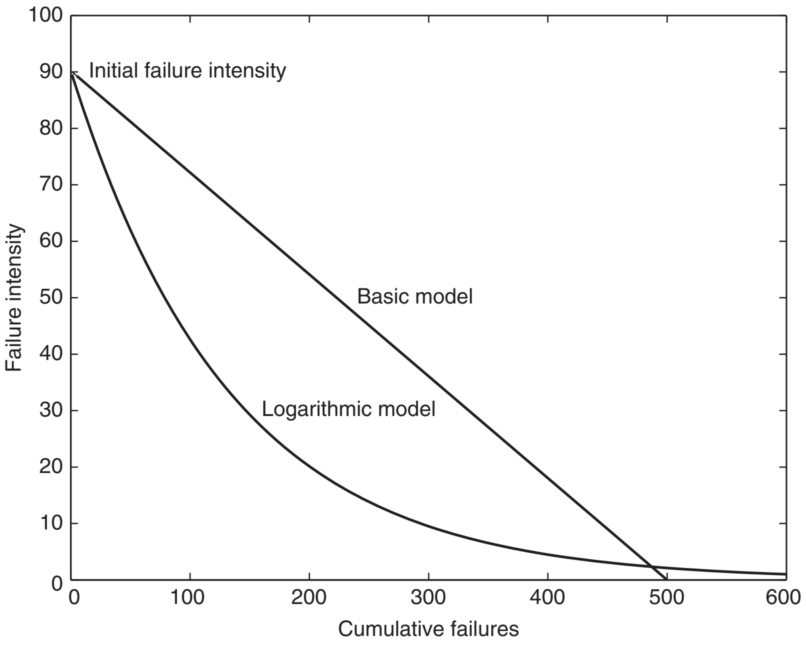

Figure 15.3 Failure intensity λ as function of cumulative failure μ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

process in the basic model. Initially, the observed failure intensity is 9 failures per unit time. The total number of failures to be observed over infinite time is assumed to be 500. When all the 500 failures have been observed and the corresponding faults have been fixed, no more system failure will be observed. Therefore, the failure intensity becomes zero at the point we have observed the final failure. The rate of decrement in failure intensity is represented by the slope of the straight line and is equal to $-\lambda_0/\nu_0 = -9/500$ per unit time. The straight line in Figure 15.3 can be expressed as follows:

$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{\nu_0}\right)$$

Since both λ and μ are functions of τ and $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we have

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

By solving the above differential equation, we have

$$\mu(\tau) = \nu_0(1 - e^{-\lambda_0\tau/\nu_0})$$

and

$$\lambda(\tau) = \lambda_0 e^{-\lambda_0\tau/\nu_0}$$

Logarithmic Model The curved line in Figure 15.3 illustrates the process of decrement in failure intensity per failure observed in the logarithmic model. Whichever reliability model chosen, the observed failure intensity remains the same. Therefore, in Figure 15.3, the initially observed failure intensity is shown to be the same, that is, 9 failures per unit time, as in the basic model. The total number of failures to be observed over infinite time is infinite. Therefore, the failure intensity never reaches zero. The relationship between failure intensity and cumulative failure count shows that faults fixed in the beginning cause a larger decrement in failure intensity than faults fixed at a later stage. This view of decrement in failure intensity is consistent with real-life systems. For example, those faults which cause a system to fail in many ways are likely to manifest in earlier failures than those faults which manifest themselves in fewer failures. Therefore, by fixing a fault early, one observes a larger drop in failure intensity than caused by fixing a fault at a later stage.

The nonlinear drop in failure intensity in the logarithmic model is captured by a decay parameter θ associated with a negative exponential function as shown in the following relationship between λ and μ:

$$\lambda(\mu) = \lambda_0 e^{-\theta\mu}$$

Since both λ and μ are functions of τ and $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we can write

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 e^{-\theta\mu(\tau)}$$

By solving the above differential equation, we obtain

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

Comparison of Reliability Models In Figure 15.4, we have plotted the expressions for $\lambda(\tau)$ in both the models and in Figure 15.5 the expressions for $\mu(\tau)$. We have assumed that the initial failure intensities are $\lambda_0 = 9$ failures per unit time in both models, the total number of failures to be observed over infinite time in the basic model is $\nu_0 = 500$ failures, and the failure intensity decay parameter $\theta = 0.0075$ in the logarithmic model.

Therefore, if we choose one of the two models to apply to a real-life software system, the next task is to estimate the model's parameters. This can be done by noting the time instants $\tau_1, \ldots, \tau_k$ at which the first k failures $\mu(\tau_1), \ldots, \mu(\tau_k)$, respectively, occur. This gives us k data points $(\tau_1, \mu(\tau_1)), \ldots, (\tau_k, \mu(\tau_k))$. We may then determine the chosen model's parameters so that the resulting plot fits the set of actually observed points. We can use a least squares technique, for example, for this purpose.

Example Assume that a software system is undergoing system-level testing. The initial failure intensity of the system was 25 failures per CPU hour, and the current

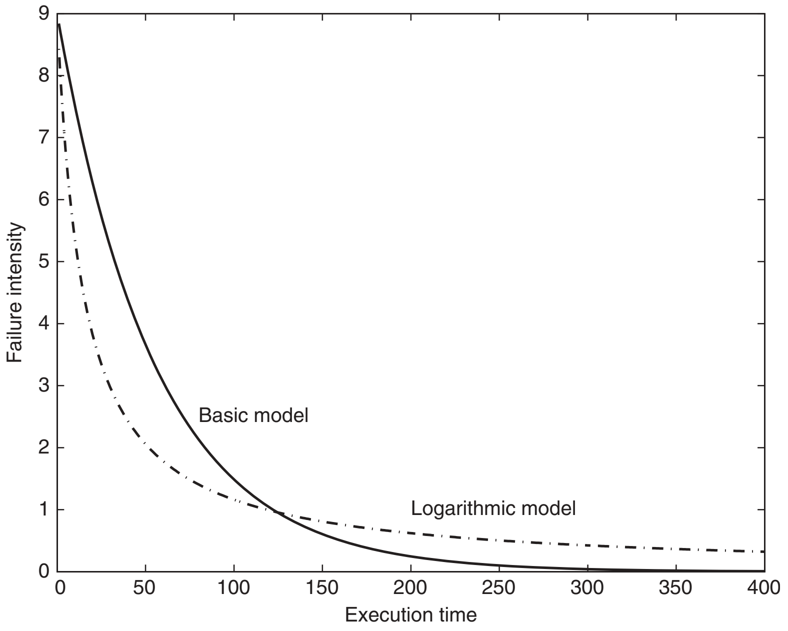

Figure 15.4 Failure intensity λ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

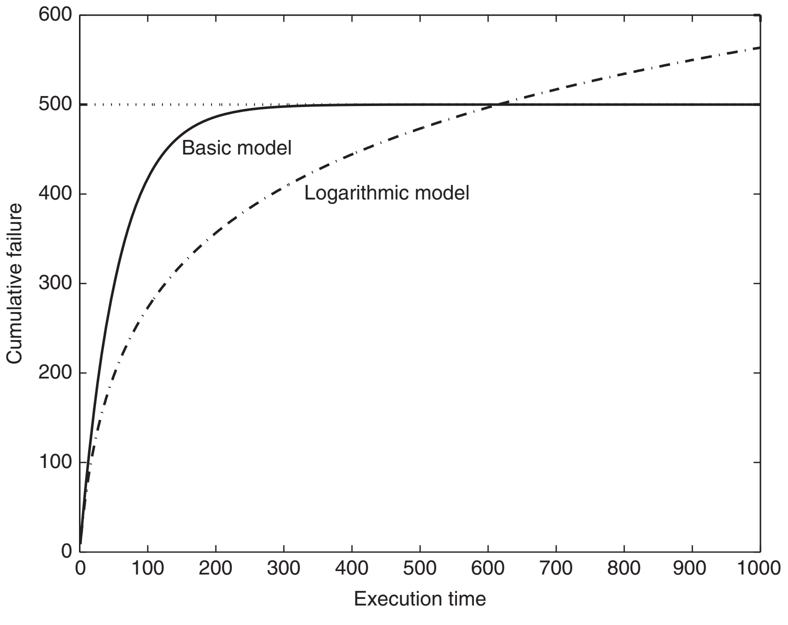

Figure 15.5 Cumulative failure μ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

failure intensity is 5 failures per CPU hour. It has been decided by the project manager that the system will be released only after the system reaches a reliability level of at most 0.001 failure per CPU hour. From experience the management team estimates that the system will experience a total of 1200 failures over infinite time. Calculate the additional length of system testing required before the system can be released.

First, an appropriate reliability model for the software system is chosen. Because of the assumption that the system will experience a total of 1200 failures over infinite time, we use the basic model.

Let us denote the current failure intensity and the desired failure intensity at the time of release as λ_c and λ_r, respectively. Assume that the current failure intensity has been achieved after executing the system for τ_c hours. Let the release time failure intensity λ_r be achieved after testing the system for a total of τ_r hours. We can write λ_c and λ_r as

$$\lambda_c = \lambda_0 e^{-\lambda_0 \tau_c/\nu_0} \qquad \lambda_r = \lambda_0 e^{-\lambda_0 \tau_r/\nu_0}$$

The quantity $\lambda_r - \lambda_c$ denotes the additional amount of system testing time needed to achieve reliability λ_r at the time of release. The quantity $\lambda_r - \lambda_c$ can be represented as follows:

$$\frac{\lambda_c}{\lambda_r} = \begin{cases} \dfrac{\lambda_0 e^{-\lambda_0 \tau_c/\nu_0}}{\lambda_0 e^{-\lambda_0 \tau_r/\nu_0}} \\ e^{(-\lambda_0 \tau_c/\nu_0)+(\lambda_0 \tau_r/\nu_0)} \\ e^{(\tau_r - \tau_c)\lambda_0/\nu_0} \end{cases}$$

or

$$\ln\left(\frac{\lambda_c}{\lambda_r}\right) = \frac{(\tau_r - \tau_c)\lambda_0}{\nu_0}$$

or

$$\tau_r - \tau_c = \begin{cases} \left(\dfrac{\nu_0}{\lambda_0}\right) \ln\left(\dfrac{\lambda_c}{\lambda_r}\right) \\ \left(\dfrac{1200}{25}\right) \ln\left(\dfrac{5}{0.001}\right) \\ 408.825 \text{ hours} \end{cases}$$

Therefore, it is required to test the system for more time so that the CPU runs for another 408.825 hours to achieve the reliability level of 0.001 failure per hour.

15.7 SUMMARY

This chapter began with an introduction to the following concepts: (i) fault and failure, (ii) execution and calendar time, (iii) time interval between failures, (iv) failures in periodic intervals, and (v) failure intensity. With these concepts in place, software reliability was defined in two ways: (i) the probability of failure-free operation of a software system for a specified time in a specified environment and (ii) failure intensity as a measure of software reliability.

Next, we discussed user's perception of software reliability, which depends upon two factors: (i) the number of faults present in the software and (ii) how the user operates the system, that is, operational profile of the system. The number of faults introduced into the system and the developer's inability to detect many of those faults are discussed, which include (i) size and complexity of code, (ii) characteristics of development process, (iii) education, experience, and training of personnel, and (iii) operational environment. In summary, software reliability is determined by a complex combination of a number of factors with wide-ranging characteristics. After that, we explained various applications of software reliability.

Then we presented the notion of operational profiles developed by Musa at Bell Laboratories. An operational profile is a quantitative characterization of how actual users operate a system. Operational profiles are represented in two alternative ways: tabular form and graphical form. Operational profiles can be used to guide the design of a user interface, the determination of where to put more resources for system development, the selection of test cases, and the development of user manuals.

Finally, two mathematical models of reliability were explained: the basic model and the logarithmic model. In these two models, failure intensity as a measure of reliability is expressed as a function of executed time.

LITERATURE REVIEW

Measuring the reliability of large software systems presents both technical and management challenges. Large software systems involve a large number of simultaneous users, say, several thousands, and most of them may be located in different geographic locations. Some examples of large software systems are software controlling the operation of cellular phone networks, on-line purchase systems, and banking systems, to name just a few. The books by Musa, Iannino, and Okumoto [7] and Musa [8] address a number of issues concerning software reliability: The technical issues addressed by Musa, Iannino, and Okumoto are as follows:

- **Parameter Determination:** Five methods of parameter determination, namely *prediction*, *estimation*, *identification*, *formula and/or experience*, and *data*, have been discussed.
- **Project-Specific Techniques:** Techniques for dealing with different special problems have been discussed. For example, they explain ways to estimate the execution time of occurrence of a failure. Another project-specific technique concerns measuring failure times in multiple installations.

The concept of operational profile has been presented in great detail by Musa (*Software Reliability Engineering*, McGraw-Hill, New York, 1999). Specifically, the main activities related to preparing an operational profile have been discussed. Moreover, the handling of the evolution of the definition of operation during system development has been discussed.

The management aspects of software reliability have been discussed in the books by both Musa, Iannino, and Okumoto [7] and Musa [8]. How to plan for implementing a reliability program has been discussed by Musa, Iannino, and Okumoto [7]. Specifically, the following activities have been explained to run a reliability program:

- **Data Collection:** Data collection for reliability analysis involves activities such as what data to gather, how to motivate data takers, how to make data collection easy, collecting data in real time, feeding results back, and recordkeeping.
- **Use of Consultants:** Consultants play an important role in technology transfer concerning reliability engineering. A consultant can be a professional external consultant or an individual inside the organization. By virtue of their expertise, consultants have a valuable impact throughout an organization.

Musa, Iannino, and Okumoto [17] present a few more models, such as Poisson-type models and binomial-type models. Lyu [19] presents several other reliability models and actual failure data from field operations of large software systems.

Another prominent method similar to reliability testing is known as *statistical testing*, which uses a formal experimental paradigm for random testing according to a usage model of the software. In statistical testing, a model is developed to characterize the population of uses of the software, and the model is used to generate a statistical correct sample of all possible uses of the software. Performance on the sample is used as a basis for conclusions about general operational reliability. Interested readers are referred to the following book and articles for discussion on this topic:

N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud, "A Machine Learning Approach for Statistical Software Testing," in *Proceedings, International Conference on Artificial Intelligence*, Hyderabad, India, Morgan Kaufman, San Francisco, January 6–12, 2007, pp. 2274–2278.

S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering*, Addison-Wesley, Reading, MA, 1999.

G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical Testing of Software Based on a Usage Model," *Software Practice and Experience*, Vol. 25, No. 1, January 1995, pp. 97–108.

J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, October 1994, pp. 812–824.