Source: STANDARD HANDBOOK OF ELECTRONIC ENGINEERING

CHAPTER 3.6
SOFTWARE RELIABILITY

Samuel Keene, Allen Nikora

INTRODUCTION AND BACKGROUND PERSPECTIVE

This chapter describes software reliability, how it can be measured, and how it can be improved. Software is a script written by a programmer dictating how a system will behave under its anticipated operating conditions (called its *main line* function). These conditions are sometimes referred to as "sunny day operations." This part of the software has been tested extensively by the time it is shipped. Problems typically arise in handling program exceptions or branching conditions when these were not properly accounted for in its original design. For example, a planetary probe satellite was programmed to switch over to its backup power system if it did not receive any communications from the ground for a 7-day period. It was assumed that such lack of communication would imply a disruption in the satellite communications power system. The strategy was to switch over to the backup power supply. This switching algorithm was built into the software. In the second decade following launch, the ground crews inadvertently went 7 days without communicating to to the probe. The satellite followed its preprogrammed instructions and switched the system over to backup power, even though the mainline system was still functioning. Unfortunately the standby system was not. So the logical error led to switching the satellite into a failed state. The satellite program was just doing what it had been told to do more than 10 years earlier.

There are many everyday analogies to software faults and failures. Any transaction having residual risks and uncertainties has a counterpart in software reliability. A real estate contract could include a flaw in the title: the house might encroach on an easement, be in a floodplain, or be flawed in its construction. Prudent parties seek to avoid such faults or their likelihood and build defenses against their occurrences. Likewise, a prudent programmer will think proactively to deal with contingent conditions and thereby mitigate any adverse effects.

Managing the programming consistencies problem is exacerbated in developing large programs. This might be considered analogous to achieving a consistent script with 200 people simultaneously writing a novel. There are many opportunities for disconnects or potential faults.

Twenty years ago, software was typically assumed to be flawless. Its reliability was considered to be 1.0. Nothing was thought to break or wear out. Hardware problems dominated systems reliability concerns. Software (and firmware) has now become ubiquitous in our systems and is most often seen as the dominant failure contributor to our systems. For example, the infamous Y2K bug has come and passed. This bug came about because programmers' thought-horizon did not properly extend into the new millenium. When the clock rolled over into the twenty-first century, software that thought the date was 1900 could disastrously fail. Fortunately, its impact was less than forecasted. Problem prevention has cost about $100B worldwide, which is significantly less than many forecasters predicted (http://www.businessweek.com/1999/02/b3611168.htm). Hopefully programmers, or information technology (IT) people, have learned from this traumatic experience. Y2K has sensitized the general public to the greater potential impact of software failures.

Imagine what might have happened if this problem caught us unawares and took its toll. Life as we know it could have been shocked. For example, disruptions could impact supermarket scanners and cash registers. Our commerce could have been foiled by credit cards not functioning.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

The use of embedded dates in software is pervasive. There are five exposure areas related to the Y2K or date related type of problem:

1. Operating system embedded dates
2. Application code embedded dates
3. Dates in the embedded software
4. Data (with potential command codes such as 9999 used as a program stop)
5. Dates embedded in interfacing commercial off the shelf (COTS) software

The Y2K software audits had to examine all the above five potential fault areas to ensure that the software was free of the date fault. This type of problem can only be avoided by fully developing the program requirements; this means defining what the program *should not do* as well as what it *should do*. The program requirements should also specify system desired behavior in the face of operational contingencies and potential anomalous program conditions. A good way to accomplish this is by performing failure modes and effects analysis (FMEA) on the high-level design (http://www.fmeainfocentre.com). This will systematically examine the program responses to aberrant input conditions.

Good program logic will execute the desired function without introducing any logical ambiguities and uncertainties. For example, some different C compilers use different order of precedence in mathematical operations. To ensure proper compiling, it is best to use nested parentheses to ensure the proper order of mathematical execution.

We now have an IT layer that surrounds our life and which we depend on. Internet use is pervasive and growing. Our offices, businesses, and personal lives depend on computer communications. We feel stranded when that service is not available, even for short periods. The most notorious outage was the 18-h outage experienced by AOL users in August 1996 (http://www.cnn.com/TECH/9608/08/aol.resumes). This resulted from an undetected operator error occurring during a software upgrade. The cause of failure was thought to be solely because of the upgrade. This confounding problem misled the analysts tracking the problem. Thus, 18 h were required to isolate and correct the problem.

More importantly, many of our common services depend on computer controllers and communications. This includes power distribution, telephone, water supply, sewage disposal, financial transactions, as well as web communications. Shocks to these services disrupt our way of life in our global economy. We are dependent on these services being up and functioning. The most critical and intertwining IT services that envelop our lives, either directly or indirectly are:

1. Nuclear power
2. Medical equipment and devices
3. Traffic control (air, train, drive-by-wire automobiles, traffic control lights)
4. Environmental impact areas (smoke stack filtration)
5. Business enterprise (paperless systems)
6. Financial systems
7. Common services (water, sewer, communications)
8. Missile and weapons systems
9. Remote systems (space, undersea, polar cap, mountain top)
10. Transportation (autos, planes, subways, elevators, trains, and so forth)

SOFTWARE RELIABILITY DEFINITIONS AND BASIC CONCEPTS

Failures can be defined as the termination of the ability of a functional unit to perform its required function. The user sees failures as a deviation of performance from the customer's requirements, expectations, and needs. For example, when a word-processing system locks up and doesn't respond, that is a failure in the eyes

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

SOFTWARE RELIABILITY

of the customer. This is a failure of commission. A second type of failure occurs when the software is incapable of performing a needed function (failure of omission). This usually results from a "breakdown" in defining software requirements, which often is the dominant source of software and system problems. The customer makes the call as to what constitutes a failure. A failure signals the presence of an underlying fault. Faults are a logical construct (omission or commission) that leads to the software failure. Hardware faults can sometimes be corrected or averted by making software changes on a fielded system. The system software changes are made when they can be done faster and cheaper than changing the hardware.

Faults are in correct system states that lead to failure. They result from *latent defects* (these defects are hidden defects, whereas *patent* defects can be directly observed). The first time the newly developed F-16 fighter plane flew south of the equator, the navigational system of the plane caused it to flip over. This part of the navigational software had never been exercised prior to this trial. In the Northern Hemisphere, there was no problem. A fault is merely a susceptibility to failure, it may never be triggered and thus a failure may never occur.

Fault triggers are those program conditions or inputs that activate a fault and instigate a failure. These failures typically result from untested branch, exception, or system conditions. Such triggers often result from unexpected inputs made to the system by the user, operator, or maintainer. For example, The user might inadvertently direct the system to do two opposing things at once such as to go up and go down simultaneously. These aberrant or off-nominal input conditions stress the code robustness. The *exception-handling* aspects of the code are more complex than the operational code since there are a greater number of possible operational scenarios. Properly managing exception conditions is critical to achieving reliable software.

Failures can also arise from a program navigating untested operational paths or sequences that prove stressful for the code. For example, some paths are susceptible to program timing failures. In these instances the desired sequence of program input conditions may violate the operational premise of the software and lead to erroneous output conditions. Timing failures are typically sensitive to the executable path taken. There are so many sequences of paths through a program that they are said to be *combinatory-explosive.* For example, 10 paths taken in random sequence lead to 2^{10} or 1024 combinations of possible paths. Practical testing considerations limit the number of path combinations that can be tested. There will always be a residual failure risk posed by the untested path combinations.

Errors are the inadvertent omissions or commissions of the programmer that allow the software to misbehave, relative to the user's needs and requirements. The relationship between errors, faults, and failures is shown in Fig. 3.6.1.

Ideally, a good software design should correctly handle all inputs under all environmental conditions. It should also correctly handle any errant or off-nominal input conditions. An example of this is can be demonstrated in the following simple arithmetic expression:

$$A + B = C \tag{1}$$

This operation is deterministic when the inputs A and B are real numbers. However, what program response is desired when one of the inputs is unbounded, missing, imaginary, or textural? That is, how should the program handle erroneous inputs? Equation (1) would function best if the program recognized the aberrant condition. Then the software would not attempt to execute the program statement shown above but it would report the errant input condition back to the user. Then this code would be handling the off-nominal condition in a robust manner. It would behave in the manner that the user would desire.

The main line code usually does its job very well. Most often, the main line code is sufficiently refined by the extensive testing it has undergone. The software failures or code breakdowns occur when the software exception code does not properly handle a rarely experienced and untested condition. More frequently the failure driver or trigger is an abnormal input or environmental conditions. An example of a rarely occurring or unanticipated condition was the potential problems posed for programs, failing to gracefully handle the millenium clock rollover. A classic example of improperly handling an off-nominal condition was the infamous

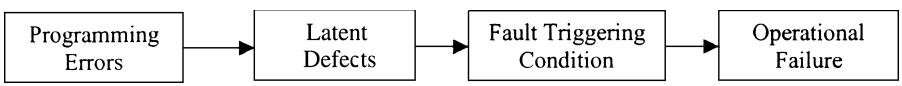

FIGURE 3.6.1 The logical path to software failures.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

"Malfunction 54" of the Therac 25 radiation therapy machine. It allowed patients to be exposed to potentially lethal radiation levels because the software controls did not satisfactorily interlock the high-energy operating system mode following a system malfunction. The operator had a work-around that allowed the radiation therapy machine to function but unknown to the operator; the work-around defeated the software interlock.

There is a software development discipline that directly addresses the proper handling of branching and exception conditions. That is the so-called "Clean Room" software design (http://www.dacs.dtic.mil/databases/url/key.hts?keycode=64). It directly ensures validating the branches and exception conditions by routinely checking the exception conditions before it performs the main line operational function. It does these checks at each operational step throughout the program.

System Management Failures

Murphy reports that the biggest cause of problems for modern software stem from either requirements' deficiencies or improperly understood system interfaces [Murp95]. These are communications-based problems, which he labels as *system management* failures. An example of this occurred when the Martian probe failed in the fall of 1999 (http://mars.jpl.nasa.gov/msp98/orbiter/). This failure was because of at least a couple of system management problems. First, there was a reported break down in communications between the propulsion engineers and the navigation scientists, at a critical point in the design cycle. They were working under different assumptions that compromised design reliability. Second, the newspaper headlines publicized another program communication problem. NASA and its contractor had unconsciously been mixing metric and British dimensional units. The resulting tower of Babel situation helped misguide the $125 million dollar space probe.

System management problems are truly communication problems. This author has been a life-long student of system failures (and successes). Today, for the most part, hardware is reliable and capable. Almost all significant system problems are traceable to a communications break down or too limited design perspective on the programmer or designer's part. Also, it needs to be recognized that programmers typically lack the "domain knowledge" of the application area for which they are designing the controlling software. For example, the programmers know the C language or JAVA but they are challenged when it comes to understanding the engineering aspects of a rocket control system. The system development people need to recognize these limitations and be proactive supporting program requirements development.

Failure Modes and Effects Analysis

A FMEA on the new software can also help to refine requirements. This analysis will help focus what the software is expected or desired to do as well as what behaviors are not desired. The software developer is first asked to explain the software's designed functionality. Then the person performing the FMEA will pose a set of abnormal input conditions that the software should handle. The software designer has been focusing on the positive path and now FMEA rounds out the remainder of the conditions that the software must accommodate. Dr. Keene has found design FMEA analysis, nominally takes two hours working directly with the software developer. This analysis has always led to the developer making changes to make the design more robust.

Code Checking and Revision

Designers must consider the possibility of off-nominal conditions that their codes must successfully handle. This is a proactive or defensive design approach. For example, in numeric intensive algorithms, there is always the possibility of divide-by-zero situations where the formula would go unbounded. When many calculations are involved or when one is dealing with inverting matrices, a divide-by-zero happens too frequently. The code can defend itself by checking the denominator to ensure that it is nonzero before dividing. There is a performance cost to do this operation. It is best when the designer is aware of the exposures and

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

TABLE 3.6.1 Some Typical Fault Tolerance Programming Considerations

1.	How will you detect a failure in your module? Programmed exception handling or abort with default exception handling?
2.	Will you detect out-of-range data?
3.	How will you detect invalid commands?
4.	How will you detect invalid addresses?
5.	What happens if bad data are delivered to your module?
6.	What will prevent a divide-by-zero from causing an abort?
7.	Will there be sufficient data in a log to trace a problem with the module?
8.	Will the module catch all its own faults? (What percentage detection and unambiguous isolation?)
9.	Do you have dependencies for fault detection and recovery on other modules or the operating system?
10.	What is the probability of isolating faults to the module? Line of code?
11.	Does your module handle "exceptions" by providing meaningful messages to allow operator-assisted recovery without total restart?

the tradeoffs. A sample list of fault-tolerance considerations that can be applied to the design process is shown in Table 3.6.1.

Code changes are best made at planned intervals to ensure proper review, inspection, integration, and testing to ensure the new code fits seamlessly into the architecture. This practice precludes "rushed" changes being forced into the code. The new code changes will be regression tested through the same waterfall development process (requirements validation, high-level code inspection, low-level code inspection, unit test, software integration and test, and system test) as the base code. This change process preserves the original code development quality and maintains traceability to the design requirements. Using prototype releases and incremental releases helps reveal and refine program requirements. The spiral development model promotes progressive software validation and requirements refinement.

SOFTWARE RELIABILITY DESIGN CONSIDERATIONS

Software reliability is determined by the quality of the development process that produced it. Reliable code is exemplified by its greater understandability. Software understandability impacts the software maintainability (and thus its reliability). The architecture will be more maintainable with reliable, fault-free code requiring minimal revision. The software developer is continually refining (maintaining) the code during development and then the field support programmer maintains the code in the field.

Code reliability is augmented by:

1. Increased cohesion of the code modules
2. Lowered coupling between modules, minimizing their interaction
3. Self describing, longer variable, mnemonic names
4. Uniform conventions, structures, naming conventions and data descriptions
5. Code modularity optimized for understandability while minimizing overall complexity
6. Software documentation formatted for greatest clarity and understanding
7. Adding code commenting to explain any extraordinary programming structures
8. Modular system architecture to provide configuration flexibility and future growth
9. Restricting a single line of code to contain only a single function or operation
10. Avoiding negative logic; especially double negative logic in the design
11. Storing any configuration parameters expected to change in a database to minimize the impacts of any changes on the design
12. Ensuring consistency of the design, source code, notation, terminology, and documentation

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

13. Maintaining the software documentation (including flow charts) to reflect the current software design level
14. Harmonizing the code after each modification to ensure all the above rules are maintained

There are also aging conditions in software that make it prone to failure. This occurs as its operating environment entropy increases. Consider the case of a distributed architecture air defense system, which had recognizance satellite receivers distributed across the country. These receivers experienced random failures in time. On one occasion the entire countrywide system was powered down and reset at one time. This system synchronization revealed an amazing thing. When the satellite receivers failed, they all failed simultaneously. Synchronizing the computer applications revealed and underlying common fault. A data buffer was tracking some particular rare transactions. It processed them and then passed them along. This register maintained a stack of all its input data, never clearing out the incoming data. So its overflow was inevitable. This overflow was a rare event but the fault was strongly revealed once the system was synchronized.

Computer System Environment

Software reliability is strongly affected by the computer system environment (e.g., the length of buffer queues, memory leaks, and resource contention). There are two lessons learned from software-aging considerations. First, software testing is most effective (likely to reveal bugs) when the distributed software are synchronously aged, i.e., restarted at the same time. To the other extreme, the distributed assets in the operational system will be more reliable if they are asynchronously restarted. Then processors are proactively restarted at advantageous intervals to reset the operating environment and restore the system entropy. This is called "software rejuvenation," or averting system "state accretion failures" (http://www.software-rejuvenation.com). The system assets will naturally age asynchronously as the software assets are restarted following random hardware maintenance activities. The software running asynchronously helps preserve the architectural redundancy of system structure where one system asset can backup others. No redundancy exists if the backup asset concurrently succumbs to the same failure as the primary asset. Diversity of the backup asset helps preclude common-mode failures.

Software and system reliability will be improved by:

1. Strong and systematic focus on requirements development, validation, and traceability with particular emphasis on system management aspects. Full requirements development also requires specifying things the system should not do as well as those desired actions. For example, heat-seeking missiles should not boomerang and return to the installation that fired them. Quality Functional Development (QFD) is one tool that helps ensure requirement completeness, as well as fully documenting the requirements development process (http://www.shef.ac.uk/~ibberson/QFD-Introduction.html).
2. Institutionalizing a "Lessons Learned" database and using it to avoid past problems and mitigating potential failures during the design process, thinking defensively, examining how the code handles off-nominal program inputs, designing to mitigate these conditions.
3. Prototype software releases are most helpful in clarifying the software's requirements. The user can see what the software will do and what it will not do. This operational prototype helps to clarify the user's needs and the developer's understanding of the user's requirements. Prototypes help the user and the developer gather experience and promote better operational and functional definition of the code. Prototypes also help clarify the environmental and system exception conditions that the code must handle. To paraphrase an old saying, "a prototype (picture) is worth a thousand words of a requirements statement," the spiral model advocates a series of prototypes to recursively refine the design requirements (http://www.sei.cmu.edu/cbs/spiral2000/february2000/Boehm).
4. Building in diagnostic capability. Systems' vulnerability is an evidence of omissions in our designs and implementation. Often we fail to include error detection and correction (EDAC) capability as high-level requirements. The U.S. electrical power system, which is managed over a grid system, is able to use its atomic clocks for EDAC. These clocks monitor time to better than a billionth of a second. This timing capability's primary purpose is for proper cost accounting of power consumption. This timing capability

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

SOFTWARE RELIABILITY

provides a significant EDAC tool to disaggregate failure events that used to appear simultaneous. These precise clocks can also time reflections in long transmission lines to locate the break in the line to within 100 m. The time stamps can also reveal which switch was the first to fail over, helping to isolate the root failure cause.

5. Performing a potential failure modes and effects analysis to harden the system response to properly deal with abnormal input conditions.
6. Failures should always be analyzed down to their root cause for repair and prevention of their reoccurrence. To be the most proactive, the system software should be parsed to see if other instances exist where this same type failure could result. The space shuttle designers did an excellent job of leveraging their failures to remove software faults. In one instance, during testing, they found a problem when system operations were suspended and then the system failed upon restarting the system. Their restart operation, in this instance, somehow violated the original design conception. The system could not handle this restart mode variation and it "crashed." This failure revealed an underlying fault structure. The space shuttle program parsed all 450 KSLOC of operational on board code. Then two other similar fault instances were identified and removed. So, in this instance, a single failure led to the removal of three design faults. This proactive fault avoidance and removal process is called the "Defect Prevention Process (DPP)." DPP was the cornerstone of the highly reliable space shuttle program (http://www.engin.umd.umich.edu/CIS/course.des.cis565/lectures/sep15.html).
7. Every common mode failure needs to be treated as critical, even if its system consequence is somehow not critical. Every thread that breaks the independence of redundant assets needs to be resolved to its root cause the remedied.
8. Studying and profiling the most significant failures. Understanding the constructs that allowed the failure to happen. Y2K is an example of a date defect. One should then ask if other potential date roll over problems existed. The answer is yes. For example, the global positioning system (GPS) clock rolled over August 22, 1999. On February 29, 2000, the Leap Year exception had to be dealt with. Here one failure can result in identifying and removing multiple faults of the same type.
9. Performing fault injection into systems, as part of system development, to speed the maturity of the software diagnostic and fault-handling capability.

Change Management

Operational reliability requires effective change management: Beware small changes can have grave consequence. All too often, small changes are not treated seriously enough. Consequently, there is significant error proneness in making small code changes.

Defect rate:	1 line	50 percent
	5 lines	75 percent
	20 lines	35 percent

The defects here are any code change that results in anomalous code behavior, i.e., changes causing the code to fail. Often, small changes are not given enough respect. They are not sufficiently analyzed or tested. For example, DSC Communications Corp., the Plano Texas Company, signaling systems were at the heart of an unusual cluster of phone outages over a two-month period of time. These disruptions followed a minor software modification. The Wall Street Journal reported, "Three tiny bits of information in a huge program that ran several million lines were set incorrectly, omitting algorithms—computation procedures—that would have stopped the communication system from becoming congested, with messages. . . . Engineers decided that because the change was minor, the massive program would not have to undergo the rigorous 13-week (regression) test that most software is put through before it is shipped to the customer." Mistake!

Post-release program changes increase the software's failure likelihood. Such changes can increase the program complexity and degrade its architectural consistency. Because credit card companies have most of their volume between November 15 and January 15 each year, they have placed restrictions on telephony changes during that time period, to limit interruptions to their business during the holiday period.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

EXECUTION-TIME SOFTWARE RELIABILITY MODELS

After the source code for a software system has been written, it undergoes testing to identify and remove defects before it is released to the customer. During this time, the failure history of the system can be used to estimate and forecast its reliability software reliability. The system's failure history as input to one or more statistical models, which produce as their output quantities related to the software's reliability. The failure history takes one of the two following forms:

- Time between subsequent failures
- Number of failures observed during each test interval, and the length of that interval

The models return a probability density function (pdf) for either the time to the next failure, or the number of failures in the next test interval. This pdf is then used to estimate and forecast the reliability of the software being tested. A brief description of three of the more widely used execution time models is given in the following sections. Further details on these and additional models may be found in [Lyu96].

Jelinski-Moranda/Shooman Model

This model, generally regarded as the first software reliability model, was published in 1972 by Jelinski and Moranda (Jeli72). This model takes input in the form of times between successive failures. The model was developed for use on a Navy software development program as well as a number of modules of the *Apollo* program. Working independently of Jelinski and Moranda, Shooman (Shoo72) published an identical model in 1972. The Jelinski-Moranda model makes the following assumptions about the software and the development process:

1. The number of defects in the code is fixed.
2. No new defects are introduced into the code through the defect correction process ("perfect debugging").
3. The number of machine instructions is essentially constant.
4. Detections of defects are independent.
5. During testing, the software is used in a similar manner as the anticipated operational usage.
6. The defect detection rate is proportional to the number of defects remaining in the software.

From the sixth assumption, the hazard rate z(t) can be written as

$$z(t) = KE_r(t)$$

- K is a proportionality constant.
- $E_r(t)$ is the number of defects remaining in the program after a testing interval of length t has elapsed, normalized with respect to the total number of instructions in the code.

The failure rate $E_r(t)$ in turn is written as

$$E_r(t) = \frac{E_T}{I_T} - E_C(t) \tag{2}$$

- E_T is the number of defects initially in the program
- I_T is the number of machine instructions in the program
- $E_c(t)$ is the cumulative number of defects repaired in the interval 0 to t, normalized by the number of machine instructions in the program.

The simple form of the hazard rate (exponentially decreasing with time, linearly decreasing with the number of defects discovered) makes reliability estimation and prediction using this model a relatively easy task.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

The only unknown parameters are K and E_T; these can be found using maximum likelihood estimation techniques.

Schneidewind Model

The idea behind the Schneidewind model [Schn75, Schn93, Schn97] is that more recent failure rates might be a better predictor of future behavior than failure rates observed in the more distant past. The failure process may be changing over time, so more recent observation of failure behavior may better model the system's present reliability. This idea is implemented as three distinct forms of the model, each form reflecting the analyst's view of the importance of the failure data as a function of time. The data used for this model are the number of observed failures per test interval, where all test intervals are of the same length. If there are n test intervals, the three forms of the model are:

1. Use all of the failure counts from the n intervals. This reflects the analyst's opinion that all of the data points are of equal importance in predicting future failure behavior.
2. Ignore completely the failures from the first s-1 test intervals. This reflects the opinion that the early test intervals contribute little or nothing in predicting future behavior. For instance, this form of the model may be used to eliminate a learning curve effect by ignoring the first few test intervals. This form of the model may also be used to eliminate the effects of changing test methods. For instance, if path coverage testing is used for the first s-1 intervals, and the testing method for the remaining intervals is chosen to maximize the coverage of data definition/usage pairs, the failure counts from the first s-1 intervals might not be relevant. In this case, the analyst might choose to use this form of the model to eliminate the failure counts form the first s-1 test intervals.
3. The cumulative failure counts from the fist s-1 intervals may be used as the first data point, and the individual failure counts from the remaining test intervals are then used for the remaining data points. This approach, intermediate between the first two forms of the model, reflects the analyst's belief that a combination of the first s-1 intervals and the remaining intervals is indicative of the failure rate during the later stages of testing. For instance, this form of the model may be used in the case where the testing method remains the same during all intervals, and failure counts are available for all intervals, but the lengths of the test intervals for the first s-1 intervals are not known as accurately as for the remaining intervals.

The assumptions specific to the Schneidewind model are as follows:

1. The cumulative number of failures by time t, $M(t)$, follows a Poisson process with mean value function $\mu(t)$. This mean value function is such that the expected number of failure occurrences for any time period is proportional to the expected number of undetected failures at that time. It is also assumed to be a bounded, nondecreasing function of time with $\lim_{f\to\infty}\mu(t) = \alpha/\beta < \infty$ where α and β are parameters of the model.

   The parameter α represents the failure rate at the beginning of interval s, and the parameter β represents the negative of the derivative of the failure rate, divided by the failure rate.
2. The failure intensity function is assumed to be an exponentially decreasing function of time.
3. The number of failures detected in each of the respective intervals are independent.
4. The defect correction rate is proportional to the number of defects remaining to be corrected.
5. The intervals over which the software is observed are all taken to be the same length. The equations for cumulative number of failures and time to the next N failures are given below.

Cumulative Number of Failures. Using maximum likelihood estimates for the model parameters a and b, with s being the starting interval for using observed failure data, the cumulative number of failures between and including intervals s and t, Fs,t is given by

$$F_{s,t} = \frac{\alpha}{\beta}\left(1 - e^{-\beta(t-s+1)}\right) \tag{3}$$

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

If the cumulative number of failures between intervals 1 and s, X_{s-1}, is added, the cumulative number of failures between intervals 1 and t is

$$F_{s,t} = \frac{\alpha}{\beta}\left(1 - e^{-\beta(t-s+1)}\right) \tag{4}$$

Time to Next N Failures and Remaining Failures. At time t, the amount of execution time, T_F (Δr, t), required to reduce the number of remaining failures by Δr is given by

$$T_F(\Delta r, t) = \frac{-1}{\beta}\log\left(1 - \left(\frac{\beta \Delta r}{\alpha}\right)e^{\beta(t-s+1)}\right) \tag{5}$$

Conversely, the reduction in remaining failure, Δr, that would be achieved by executing the software for an additional amount of time T_F is

$$\Delta r(T_F, t) = \frac{\alpha}{\beta}e^{-\beta(t-s+1)}(1 - e^{-\beta T_F}) \tag{6}$$

The Schneidewind model provides a method of determining the optimal value of the starting interval s. The first step in identifying the optimal value of s (s^*) is to estimate the parameters α and β for each value of s in the range [1, t] where convergence can be obtained [Schn93]. Then the *mean square error* (MSE) criterion is used to select s^*, the failure count interval that corresponds to the minimum MSE between predicted and actual failure counts (MSE_F), *time to next failure* (MSE_T), or *remaining failures* (MSE_r), depending on the type of prediction. The first two were reported in [Schn93]—for brevity, they are not shown here. MSE_r, developed in [Schn97], is described below. MSE_r is also the criterion for *maximum failures* ($F(\infty)$) and *total test time* (t_t) because the two are functionally related to *remaining failures* ($r(t)$); see Eqs. (9) and (13) in [Schn97] for details. Once α, β, and s are estimated from observed counts of failures, the foregoing predictions can be made. The reason MSE is used to evaluate which triple (α, β, s) is best in the range [1, t] is that research has shown that because the product and process change over the life of the software, old failure data (i.e., $s = 1$) are not as representative of the current state of the product and process as the more recent failure data (i.e., $s > 1$) [Schn93].

Although we can never know whether additional failures may occur, nevertheless we can form the difference between the following two equations for $r(t)$, the number of failures remaining at time t:

$$r(t) = \frac{\alpha}{\beta} - X_{s,t} \tag{7}$$

where $X_{s,t}$ is the cumulative number of failures observed between the including intervals s and t.

$$r(t_t) = \frac{\alpha}{\beta}e^{-\beta(t_t - s+1)} \tag{8}$$

where t_t represents the cumulative test time for the system.

We form the difference between Eq. (7), which is a function of predicted *maximum failures* and the observed failures, and Eq. (8), which is a function of *total test time*, and apply the MSE criterion. This yields the following MSE_r criterion for number of *remaining failures*:

$$MSE_r = \frac{\sum_{i=s}^{t}[F(i) - X_i]^2}{t - s + 1} \tag{9}$$

The Schneidewind model has been used extensively to evaluate the Primary Avionics Software System (PASS) for the Space Transportation System [Schn92] with very good success, especially employing the procedure for determining the optimal starting interval to obtain better fits to the data.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

Musa-Okumoto Logarithmic Poisson Model

The Musa-Okumoto model has been found to be especially applicable when the testing is done according to a nonuniform operational profile. This model takes input in the form of time between successive failures. In this model, early defect corrections have a larger impact on the failure intensity than latter corrections. The failure intensity function tends to be convex with decreasing slope for this situation. The assumptions of this model are:

1. The software is operated in a similar manner as the anticipated operational usage.
2. The detections of defects are independent.
3. The expected number of defects is a logarithmic function of time.
4. The failure intensity decreases exponentially with the expected failures experienced.
5. The number of software failures has no upper bound.

In this model, the failure intensity $\lambda(\tau)$ is an exponentially decreasing function of time:

$$\lambda(\tau) = \lambda_0 e^{-\theta\mu(\tau)} \tag{10}$$

- τ = execution time elapsed since the start of test
- λ_0 = initial failure intensity
- $\lambda(\tau)$ = failure intensity at time τ
- θ = failure intensity decay parameter
- $\mu(\tau)$ = expected number of failures at time τ

The expected cumulative number of failures at time τ, $\mu(\tau)$, can be derived from the expression for failure intensity. Recalling that the failure intensity is the time derivative of the expected number of failures, the following differential equation relating these two quantities can be written as

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 e^{-\theta\mu(\tau)} \tag{11}$$

Nothing that the mean number of defects at $\tau = 0$ is zero, the solution to this differential equation is

$$\mu(\tau) = \frac{1}{\theta}\ln(\lambda_o\theta\tau + 1) \tag{12}$$

The reliability of the program, $R(\tau_i' | \tau_{i-1})$ is written as

$$R(\tau_i' | \tau_{i-1}) = \left[\frac{\lambda_0\theta\tau_{i-1} + 1}{\lambda_0\theta(\tau_{i-1} + \tau_i')}\right]^{1/\theta} \tag{13}$$

- τ_{i-1} is the cumulative time elapsed by the time the $(i-1)$th failure is observed
- τ_i' is the cumulative time by which the ith failure would be observed

The mean time of failure (MTTF) is defined only if the decay parameter is greater than 1. According to [Musa87], this is generally the case for actual development efforts. The MTTF, $\Theta[\tau_{i-1}]$, is given by

$$\Theta[\tau_{i-1}] = \frac{\theta}{1-\theta}(\lambda_0\theta\tau_{i-1} + 1)^{1-1/\theta} \tag{14}$$

Further details of this model can be found in [Musa87] and [Lyu96].

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

Benefits of Execution Time Models. There are three major areas in which advantage can be gained by the use of software reliability models. These are planning and scheduling, risk assessment, and technology evaluation. These areas are briefly discussed below.

Planning and Scheduling

- *Determine when a reliability goal has been achieved.* If a reliability requirement has been set earlier in the development process, the outputs of a reliability model can be used to produce an estimate of the system's current reliability. This estimate can then be compared to the reliability requirement to determine whether or not that requirement has been met to within a specified confidence interval. This presupposes that reliability requirements have been set during the design and implementation phases of the development.
- *Control application of test resources.* Since reliability models allow predictions of future reliability as well as estimates of current reliability to be made, practitioners can use the modeling results to determine the amount of time that will be needed to achieve a specific reliability requirement. This is done by determining the difference between the current reliability estimate and the required reliability, and using the selected model to compute the amount of additional testing time required to achieve the requirement. This amount of time can then be translated into the amount of testing resources that will be needed.
- *Determine a release date for the software.* Since reliability models can be used to predict the additional amount of testing time that will be required to achieve a reliability goal, a release date for the system can be easily determined.
- *Evaluate status during the test phase.* Obviously, reliability measurement can be used to determine whether the testing activities are increasing the reliability of the software by monitoring the failure/hazard rates. If the time between failures or failure frequency starts deviating significantly from predictions made by the model after a large enough number of failures have been observed (empirical evidence suggests that this often occurs one-third of the way through the testing effort), this can be used to identify problems in the testing effort. For instance, if the decrease in failure intensity has been continuous over a sustained period of time, and then suddenly decreases in a discontinuous manner, this would indicate that for some reason, the efficiency of the testing staff in detecting defects has decreased. Possible causes would include decreased performance in or unavailability of test equipment, large-scale staff changes, test staff reduction, or unplanned absences of experienced test staff. It would then be up to line and project management to determine the cause(s) of the change in failure behavior and determine proper corrective action.

Likewise, if the failure intensity were to suddenly rise after a period of consistent decrease, this could indicate either an increase in testing efficiency or other problems with the development effort. Possible causes would include large-scale changes to the software after testing had started, replacement of less experienced testing staff with more experienced personnel, higher testing equipment throughput, greater availability of test equipment, or changes in the testing approach. As above, the cause(s) of the change in failure behavior would have to be identified and proper corrective action determined by more detailed investigation. Changes to the failure rate would only indicate that one or more of these causes might be operating.

Risk Assessment. Software reliability models can be used to assess the risk of releasing the system at a chosen time during the test phase. As noted, reliability models can predict the additional testing time required to achieve a reliability requirement. This testing time can be compared to the actual resources (schedule and budget) available. If the available resources are not sufficient to achieve the reliability requirement, the reliability model can be used to determine to what extent the predicted reliability will differ from the reliability requirement if no further resources are allocated. These results can then be used to decide whether further testing resources should be allocated, or whether the system can be released to operational usage with a lower reliability.

Technology Evaluation. Finally, software reliability models can be used to assess the impact of new technologies on the development process. To do this, however, it is first necessary to have a well-documented history of previous projects and their reliability behavior during test. The idea of assessing the impact of new technology is quite simple—a project incorporating new technology is monitored through the testing and operational phases using software reliability modeling techniques. The results of the modeling effort are then compared to the failure behavior of similar historical projects. By comparing the reliability measurements, it is possible to see if the new technology results in higher or lower failure rates, makes it easier or more difficult to detect failures in the software, and requires more or fewer testing resources to achieve the same reliability as the historical projects. This analysis can be performed for different types of development efforts to identify

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

those for which the new technology appears to be particularly well or particularly badly suited. The results of this analysis can then be used to determine whether the technology being evaluated should be incorporated into future projects.

Execution Time Model Limitations

The limitations of execution time models are discussed below. Briefly, these limitations have to do with:

1. Applicability of the model assumptions
2. Availability of required data
3. Determining the most appropriate model
4. The life-cycle phases during which the models can be applied.

Applicability of Assumptions

Generally, the assumptions made by execution time models are made to cast the models into a mathematically tractable form. However, there may be situations in which the assumptions for a particular model or models do not apply to a development effort. In the following paragraphs, specific model assumptions are listed and the effects they may have on the accuracy of reliability estimates are described.

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

One way of dealing with this issue is the concept of the testing compression factor [Musa87]. The testing compression factor is simply the ratio of the time it would take to cover the equivalence classes of the input space of a software system in normal operations to the amount of time it would to cover those equivalence classes by testing. If the testing compression factor can be established, it can be used to predict reliability and reliability-related measures during operations. For instance, with a testing compression factor of 10, a failure intensity of one failure per 10 h measured during testing is equivalent to one failure for every 100 h during operations. Since test cases are usually designed to cover the input space as efficiently as possible, it will usually be the case that the testing compression factor is greater than 1. To determine the testing compression factor, of course, it is necessary to have a good estimate of the system's operational profile (the frequency distribution of the different input equivalence classes) from which the expected amount of time to cover the input space during the operational phase can be computed.

- *There is an upper limit to the number of failures that will be observed during testing.* Many of the more widely used execution time models make this assumption. Models making this assumption should not be applied to development efforts during which the software version being tested is simultaneously undergoing significant changes (e.g., 20 percent or more of the existing code is being changed, or the amount of code is increasing by 20 percent or more). However, if the major source of change to the software during test is the correction process, and if the corrections made do not add a significant amount of the new functionality or behavior, it is generally safe to make this assumption. This would tend to limit application of models making this assumption to subsystem-level integration or later testing phases.
- *No new defects are introduced into the code during the correction process.* Although there is always the possibility of introducing new defects during the defect removal process, many models make this assumption to simplify the reliability calculations. In many development efforts, the introduction of new defects during correction tends to be a minor effect, and is often reflected in a small readjustment of the values of the model parameters. In [Lyu91], several models making this assumption performed quite well over the data sets used for model evaluation. If the volume of software, measured in source lines of code, being

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

changed during correction is not a significant fraction of the volume of the entire program, and if the effects of repairs tend to be limited to the areas in which the corrections are made, it is generally safe to make this assumption.

- *Detections of defects are independent of one another*. All execution time models make this assumption, although it is not necessarily valid. Indeed, there is evidence that detections of defects occur in groups, and that there are some dependencies in detecting defects. The reason for this assumption is that it enormously simplifies the estimation of model parameters. Determining the maximum likelihood estimator for a model parameter requires the computation of a joint probability density function (pdf) involving all of the observed events. The assumption of independence allows this joint pdf to be computed as the product of the individual pdfs for each observation, keeping the computational requirements for parameter estimation within practical limits.

Availability of Required Data

Most software reliability models require input in the form of time between successive failures. These data are often difficult to collect accurately. Inaccurate data collection reduces the usefulness of model predictions. For instance, the noise may be great enough that the model predictions do not fit the data well as measured by traditional goodness-of-fit tests. In some cases, the data may be so noisy that it is impossible to obtain estimates for the model's parameters. Although more accurate predictions can be obtained using data in this form [Musa87], many software development efforts do not track these data accurately.

As previously mentioned, some models have been formulated to take input in the form of a sequence of pairs, in which each pair has the form of (number of failures per test interval, test interval length). For the study reported in [Lyu91, Lyu91a, Lyu91b], all of the failure data was available in this form. The authors' experience indicates that more software development efforts would have this type of information readily available, since they have tended to track the following data during testing:

1. Date and time at which a failure was observed.
2. Starting and ending times for each test interval, found in test logs that each tester is required to maintain.
3. Identity of the software component tested during each test interval.

With these three data items, the number of failures per test interval and the length of each test interval can be determined. Using the third data item, the reliability of each software component can be modeled separately, and the overall reliability of the system can be determined by constructing a reliability block diagram. Of these three items, the starting and ending times of test intervals may not be systematically recorded, although there is often a project requirement that such logs be maintained. Under schedule pressures, however, the test staff may not always maintain the test logs, and a project's enforcement of this requirement may not be sufficiently rigorous to ensure accurate test log entries.

Even if a rigorous data collection mechanism is set up to collect the required information, there appear to be two other limitations to failure history data.

- It is not always possible to determine when a failure has occurred. There may be a chain of events such that a particular component of the system fails, causing others to fail at a later time (perhaps hours or even days later), finally resulting in a user's observation that the system is no longer operating as expected. Individuals responsible for the maintenance of the Space Transportation System (STS) and Primary Avionics Software System have reported in private discussions with colleagues of the authors several occurrences of this type of latency. This raises the possibility that even the most carefully collected set of failure history has a noise component of unknown, and possibly large, magnitude.
- Not all failures are observed. Again, discussions with individuals associated with maintaining the STS flight software have included reports of failures that occurred and were not observed because none of the STS crew was looking at the display on which the failure behavior occurred. Only extensive analysis of post-flight telemetry revealed these previously unobserved failures. There is no reason to expect that this would not occur in the operation of other software systems. This describes another possible source of noise in even the most carefully collected set of failure data.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

Identifying the Most Appropriate Model

At the time the first software reliability models were published, it was thought that there would be a refinement process leading to at most a few models that would be applicable to most software development efforts [Abde86]. Unfortunately, this has not happened—since that time, several dozen models have appeared in the literature, and we appear to be no closer to producing a generally applicable model. For any given development effort, it does not appear to be possible to determine a priori which model will be the most applicable. Fortunately, this difficulty can be mitigated by applying several different models to a single set of failure history data, and then using the methods described in [Abd86] to choose the most appropriate model results. The methods include:

1. *Determination of how much more likely it is that one model will produce more accurate results than another. Prequential likelihood* functions are formed for each model. Unlike likelihood functions, these are not used to find model parameters—rather, the values of the model parameters are substituted into the prequential likelihood functions, as are the observed times between failures, to provide numerical values that can be used to compare two or more models. The larger the value, the more likely it is that the model will provide accurate predictions. Given two models, A and B, the ratio of the prequential likelihood values, PL_A/PL_B, specifies how much more likely it is that model. A will produce accurate predictions than model B.
2. *Determination of whether a model makes biased predictions*. A model may exhibit bias by making predictions of times between failures that are consistently longer than those that are actually observed, or making predictions that are shorter than what's observed. In either case, [Abde86] describes how to draw a *u-plot* that will identify those models making biased predictions. Given two models that have the same prequential likelihood value, the model exhibiting the most bias should be discarded.
3. *Determination of whether a model's bias changes over time.* If a model exhibits bias, that bias may change over time. For instance, early in a testing effort, a model may consistently predict times between failures that are longer than what is actually observed. As testing progresses, the model's bias may change to making predicting shorter times between failures than those actually observed. Such a change in a model's bias may not show up in a u-plot, since the computations involved in producing u-plot does not preserve temporal information associated with the failure history. However, a series of further transformations allow the construction of a *y-plot*, which identifies models whose bias changes over time. Given two models with identical prequential likelihood values and identical u-plots, the model exhibiting a change in bias over time should be rejected over the one that shows no change in its bias.

Applicable Development Phases

Perhaps the greatest limitation of the execution time software reliability models is that they can only be used during the testing phases of a development effort. In addition, they usually cannot be used during unit test, since the number of failures found in each unit will not be large enough to make meaningful estimates for model parameters. These techniques, then, are useful as a management tool for estimating and controlling the resources for the testing phases. However, the models do not include any product or process characteristics that could be used to make tradeoffs between development methods, budget, and reliability.

PREDICTING SOFTWARE RELIABILITY

Software's design reliability is typically measured in terms of faults per thousand lines of source code (KSLOC). The lines of code are the executable lines of code, excluding the commentary lines of source code. Typical fault density levels, as a function of the development process, are shown in Table 3.6.2.

Code is necessarily released with faults in it. These are only estimated faults and not known faults. If they were known faults, the developer would remove them. This author has heard a hardware developer decry the notion of releasing code with faults. This critic claimed that a hardware developer would never do that. The truth is, with software one cannot test or anticipate all the scenarios that the software will have to handle. Some of these bugs

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

TABLE 3.6.2 Code Defect Rates (Faults/KSLOC)

	Total development	At delivery
Traditional development Bottom-up design, unstructured code, removal through testing	50–60	15–18
Modern practice Top-down design, structured code, design and code inspections, incremental releases	20–40	2–4
Future improvements Advanced software engineering, verification practice, reliability measurement	6–20	<1
Space shuttle code	6–12	0.5*

*The defect density of the space shuttle code is often shown as 0.1 defects/KSLOC. From private correspondence with the software quality lead engineer on the space shuttle, the author has learned that the estimated defect rate is 0.5 at turnover to the NASA customer. The space shuttle code only gets to 0.1 defects/KSLOC after eight additional months of NASA testing.

(another name for faults) will never affect a particular user. Bugs are potential problems in the code that have to be triggered by a set of special conditions. For instance, there could be a calculation in the software having a term of $1/(Z–225)$ that "explodes" or goes to infinity if $Z = 225$. For all other values this term is well behaved.

There was a lot of hype over the thousands of bugs released with Windows 2000 (http://home.rochester.rr.com/seamans). That article states, "Well, it's been 2 months since the release of Windows 2000. Where are we now? After the hype of 63,000 bugs, we've seen a few bugs creep up here and there. Many argue over whether the current bugs out there are major or minor in nature. It seems that a vast amount of the bugs found to date are minor in nature and the ones that may be major (i.e., Windows 2000 doesn't support more than 51 IP addresses) affect only a very few IT operations. In all, I think Windows 2000 has fared rather well considering what has taken place over the last year."

Keene Process Based Reliability Model

It is often required to estimate the software's reliability before the code is even developed. The Keene software reliability model projects fault density based on empirical data correlating fault density to the process capability of the underlying development process. Process capability measurement is most conveniently accomplished using the development organization's Capability Maturity Model (CMM) level, which is explained in the next paragraph. The fault density is projected based on the measured process capability. The Keene model next transforms the latent fault density into an exponential reliability growth curve over time, as failures surface and the underlying faults are found and removed. Thus, failures are traced to the underlying faults and removed. Failures occur and these are the outward manifestation of the underlying fault. The faults are removed over time through software maintenance actions and the corresponding software failure rate diminishes accordingly.

The Software Engineering Institute (SEI) has developed a CMM that is a framework describing the key elements of an effective software development process. It covers key practices for planning, engineering, management, development, assurance, and maintenance that, when followed, should improve the ability of an organization to meet cost, schedule, and quality goals. They also provide five levels of grading which permits good differentiation among process capabilities.

- SEI Level 1 (Initial): Ad hoc process development dependent on individuals. Cost, schedule, and quality are unpredictable. Most companies fall into this category.
- SEI Level 2 (Repeatable): Policies for managing software are in place. Planning is based on prior experience. Planning and tracking can be repeated. Seventy percent of companies are rated at SEI level II or I.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

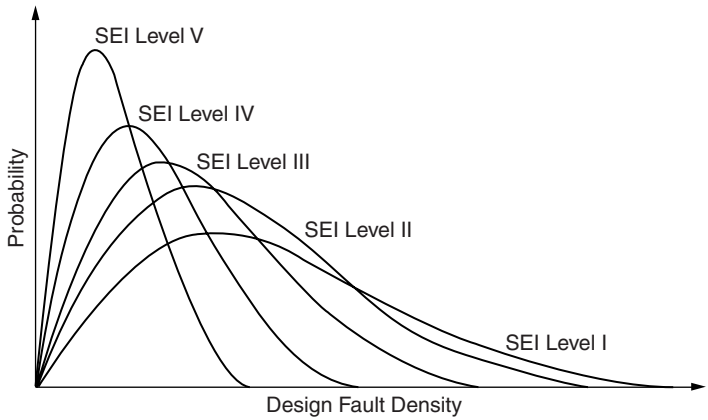

FIGURE 3.6.2 Mapping of the SEI CMM levels versus fault density distributions.

- SEI Level 3 (Defined): The process for developing the software is documented and integrated coherently. The process is stable and repeatable and the activities of the process are understood organization wide.
- SEI Level 4 (Managed): The process is measured and operates within measured limits. The organization is able to predict trends and produce products with predictable quality.
- SEI Level 5 (Optimization): The focus is on continuous improvement. The organization can identify weaknesses in the process and is proactive in preventing faults. The organization is also innovative throughout. It demonstrates a passion to find and eliminate bugs. This organization also is continually improving its development process to reduce the likelihood of bug recurrence. It learns from the errors it finds. It is dedicated to continuous measurable improvement, sometimes referred to as CMI.

Figure 3.6.2 illustrates how the SEI's CMM levels correlate with the design fault density at product shipment as shown below.

The higher capability levels are expected to exhibit lower fault density than the lower capability levels. The higher SEI process levels are indeed more capable. The higher levels are also shown to have less variability, i.e., they are a more consistent predictor of fault density. The figure above depicts that the SEI level is a good predictor (causative factor) of fault density.

The process capability level is a good predictor of the latent fault content shipped with the code. The better the process, the better the process capability ratings and the better the code developed under that process. The projection of fault density according to the corresponding SEI level is now shown in Table 3.6.3. Keene bases these fault density settings on information compiled over a variety of programs. This information is proprietary and is not included in this paper. The SEI Level 5 experience is the most publicized on the space shuttle program of 0.5 defects/KSLOC. The table shows the posited relationship between the development organization's SEI's CMM level and the projected latent fault density of the code it ships.

The software reliability model Eq. (15), shows that the fault content decreasing exponentially over time. The exponential fault reduction model follows from assuming that the number of faults being discovered and removed from the code is proportional to the total number of faults existing in the code, at any point in time. This premise is intuitively appealing and supported by empirical evidence. This relationship expressed mathematically is

$$F(t) = F_0 e^{-kt} \tag{15}$$

where $F(t)$ = current number of faults remaining in the code after calendar time t
F_0 = initial number of faults in the code at delivery, which is a function of the developer's SEI development capability
k = constant of proportionality
t = calendar time in months

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

TABLE 3.6.3 Software's Projected Latent Fault Density as a Function of CMM Level

SEI'S CMM Level	Maturity profile June 2001 1018 Organizations*	Latent fault density (Faults/KSLOC-all severitíes)	Defect plateau level for 48 months after initial delivery or 24 months following subsequent deliveries
5	Optimizing: 4.8 percent	0.5	1.5 percent
4	Managed: 5.6 percent	1.0	3.0 percent
3	Defined: 23.4 percent	2.0	5.0 percent
2	Repeatable: 39.1 percent	3.0	7.0 percent
1	Initial: 27.1 percent	5.0	10.0 percent
unrated	The remainder of companies	>5.0	not estimated

*http://www.sei.cmu.edu/sema/pdf/2001aug.pdf

The proportionality constant k is set so the latent fault content drops to a certain percentage of the initial fault discovery rate after a code-stabilizing period of time. One analysis showed field data to stabilize at a 10 percent F_0 level after 48 months [Keen94]. This 10 percent level is appropriate for a CMM Level 1 process. The higher level processes drive the stabilization level lower than 10 percent that is used here. The constant k is determined by solving Eq. (15) for $F(t)$ dropping to 10 percent at 48 months. That analysis shows to be equal 0.048 per months.

Faults are revealed by failures. The failures are then traced to root cause, removed and the fault count goes down correspondingly. So failure the software failure rate goes down in proportion to the number of faults remaining in the code.

For purposes of estimating the operational failure rate of the fielded code, one can apply a linear, piecewise approximation to the exponential fault removal profile. This simplifies calculation of an average failure rate over the time interval. The number of faults found and removed is indicative of the failure rate of the software. That is, it usually takes a failure to reveal an underlying fault in the software. The more failures that occur, the more underlying faults there are uncovered in the code. The software failure rate λ is related to the number of underlying fault removal over time as follows:

$$\lambda(t) \approx \frac{F(t_2) - F(t_1)}{t_2 - t_1} \tag{16}$$

.

Adding a constant of proportionality ρ,

$$\lambda(t) = \rho \frac{F(t_2) - F(t_1)}{t_2 - t_1} \tag{17}$$

where ρ represents a coverage factor. This is the total number of failure occurrences that a fault manifests before it is removed versus fielded population of systems. The coverage factor is made up of three components:

1. Percentage fault activation is that percent of the population likely to experience a given fault on the average. This factor is a small percentage when there are many copies in a widely distributed system such as an operating system.
2. Fault latency is measured in terms of the number of times that a unit in the failing subpopulation is likely to experience a failure before the failure is analyzed to root cause level and the corrective software patch installed.
3. Percentage critical faults that will cause the system to fail for the user.

The coverage factor is illustrated by considering 10 fielded copies of a software package, where five of the users experience problems. On average these five users experience three failures each before the problem is resolved and the underlying fault removed. The user in this example considers all failures critical. Then the

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

coverage factor would be (5 failures/10 fielded software instances) × (3 repeats) × (100 percent critical failures) over the install base of 10 users. Thus, $\rho = 1.5$. This number is based on the developer's history or by direct estimation of field performance parameters. "Measurement is important, otherwise you are just practicing."

Sometimes a single failure can lead to the removal of several faults. On the space shuttle for instance, a particular problem was experienced when reentering the code, having once exited it. This problem was called the "multi-pass problem." The problem was discovered and resolved. The entire shuttle code was searched looking for other instances of this problem. A single failure led to the correction of three different faults in the code. More often, it takes several failure occurrences before the underlying fault is found and removed.

The failure replication of the same problem will be somewhat offset by failures not being experienced across all systems. That is, the early adopters will first experience the software faults. There is also a subset of all users that stress their systems harder with greater job loads or more diverse applications. Some problems will be found and corrected before the majority of the remaining users have experienced it. This reduces the ρ factor or fault coverage. Chillarege's data show software improvement over time that was slightly steeper than an exponential growth profile (Ram Chillarege, Shriram Biyani, Jeanette Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," Fault Tolerant Computing Symposium, 1995, pp. 424–433). The customer base he reported approached a million users, so there would be plenty of early adapters to find and fix the problems before the majority of users would see them.

Rayleigh Development Model

The defect rates found in the various development phases will successfully predict the latent defects shipped with the product, (from "Metrics and Models in Software Quality Engineering," Chap. 7, by Stephen H. Kan, Addison-Wesley, 1995).

The Rayleigh model is shown below:

- CDF: $F(t) = 1 - e^{(-t/c)2}$
- PDF: $f(t) = (2/t)(t/c)^2 e^{(-t/c)2}$

The Rayleigh model is the best tool to predict software reliability during development, prior to system testing. The Rayleigh model is a member of the Weibull distribution with its shape parameter, $m = 2$. The Rayleigh prediction model forward chains all the defect discovery data, collected throughout the development phases, to predict the software's latent fault density. (Latent errors are those errors that are not apparent at delivery but manifest themselves during subsequent operations. This is contrasted to patent errors, which are obvious by inspection.) The inputs to the Rayleigh model are the defect discovery rates found in the following design phases: high level design, low level design, code and unit test, software integration and test, unit test, and system test.

There are two basic assumptions associated with the Rayleigh model. First is that the defect rate observed during the development process is positively correlated with the defect rate in the field. The higher the Rayleigh curve of defects observed during development, the higher the expected latent defect rate of the released code. This is related to the concept of error injection. Second is that given the same error injection rate, if more defects are discovered and removed earlier, fewer will remain in later stages. The output of the Rayleigh model is the expected latent fault density in the code when it is released.

The following are priority definitions used for quantifying development defects. The Rayleigh analysis can be used to project any or all of these priority defects itemized below:

- Critical (Priority 1): An error which prevents the accomplishment of an operational or mission essential function in accordance with specification requirements (e.g., causes a computer program stop), which interferes with use of the system to the extent that it prevents the accomplishment of a mission essential function, or which jeopardizes personnel safety.
- Major (Priority 2): An error which adversely affects the accomplishment of a mission essential function in accordance with specification requirements so as to degrade performance of the system and for which no alternative is provided. Rebooting or restarting the application is not an acceptable alternative, as it constitutes unacceptable interference with use of the system.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

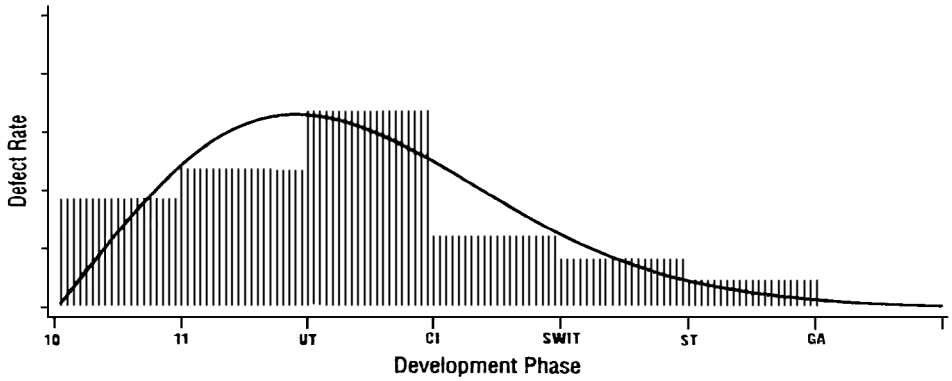

FIGURE 3.6.3 Rayleigh defect removal profile curve.

- Minor (Priority 3): An error which adversely affects the accomplishment of an operational or mission essential function in accordance with specification requirements so as to degrade performance and for which there is a reasonable alternative provided. Rebooting or restarting the application is not an acceptable alternative, as it constitutes unacceptable interference with use of the system.
- Annoyance (Priority 4): An error, which is an operator inconvenience or annoyance and does not affect an operational or mission essential function.
- Other (Priority 5): All others errors.

The difference between the amount of faults injected into the code and those that escape detection and end up being shipped to the field reflects the goodness of the development process. Defects are removed by a number of development quality processes such as inspections, reviews, and tests. Eight-five percent of all the injected faults were removed during the development phase prior, to the system test, on the highly successful space shuttle program. The better the development process, the sooner the defects are removed during the development process and the fewer defects that escape (detection and removal) and go to the field as latent defects. Once the code is in operational test or the field, the computer aided software reliability estimation (CASRE) tool can be applied to measure its operational reliability.

CASRE

Computer aided software reliability estimation, developed with funding from the U.S. Air Force Operational Test and Evaluation Center (AFOTEC), is a stand-alone Windows-based tool that implements 10 of the more widely used execution time software reliability models. CASRE's goal was to make it easier for nonspecialists in software reliability to apply models to a set of failure history data, and make predictions of what the software's reliability will be in the future. In 1992, when the first version was released, it was the first software reliability modeling tool that displayed its results as high-resolution plots and employed pull-down menus in its user interface—other tools at that time interacted with the user via command lines or batch files, and displayed their results in tabular form. It also allows users to define combination models [Lyu91a] to increase the predictive accuracy of the models. In addition, it implements the model selection criteria described in [Abde86] to allow user to choose the most appropriate model results from a set of results. Finally, it implements trend tests that can be applied to the failure data to help the user determine whether it's even appropriate to apply reliability models to the data.

The modeling and model applicability analysis capabilities of CASRE are provided by Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS), a software reliability modeling tool developed by Dr. William H. Farr at the Naval Surface Warfare Center, Dahlgren, Virginia. In implementing CASRE, the

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

SOFTWARE RELIABILITY

original SMERFS user interface and the SMERFS modeling libraries were linked into the user interface developed for CASRE. The combination modeling and trend test capabilities, however, are new for CASRE, as are the model ranking capabilities.

Version 3.0 of CASRE is available from the Open Channel Foundation at http://www.openchannelfoundation.org.

The following steps illustrate a typical application of CASRE to a set of failure data:

1. When a set of failure data is first opened with CASRE, the data are displayed both in text and graphical form. One window displays the text of the failure data, and the other displays a high-resolution plot of the data. CASRE can use two types of failure data—either time between subsequent failures, or the number of failures observed in a test interval of a given length. If the data are in the form of times between subsequent failures, the window displaying text shows the failure number, the elapsed time since the previous failure, and the failure's severity on an arbitrary scale of 0 to 9. The initial plot shows the times between subsequent failures as a function of failure number. For interval data, the text display shows the interval number, the number of failures observed in that test interval, the length of the interval, and the severity of the failures in that interval. The initial plot for this type of data shows the number of failures per test interval as a function of test interval number.
2. Whether the reliability of the software being analyzed is increasing depends on many factors, such as the type of testing being performed, and whether substantial amounts of the software's functionality are being added or modified. To determine the advisability of applying reliability models to the failure data, trend tests should be applied to determine whether the failure data are exhibiting reliability growth. If these trend tests indicate that the failure data exhibit reliability growth, then the user can be confident in applying a software reliability model to the data. However, it may be the case that failures are detected at a nondecreasing, or even an increasing rate, in which case the reliability of the software being analyzed would actually be decreasing. As mentioned above, this could happen if significant amounts of additional functionality are being added while testing is taking place. In this case, application of a reliability model would not be advisable, since all of the models assume that the reliability of the software increases as testing progresses. CASRE implements two trend tests to help users determine whether the data are exhibiting reliability growth—a running arithmetic average, and the Laplace test [Lyu96].
3. If the trend tests indicate it is advisable to apply reliability models to the failure data, the next step is the selection of the models to be used. Because it is generally not possible to determine a priori the most appropriate model for a set of data, it is recommended that several different models be applied to the same set of data and the methods described in [Abde86] be applied to the results to select the most appropriate model. CASRE allows users to apply more than one model to the current set of failure data. Once the models have completed, users can select up to three sets of model results, and plot them in several different ways. Among these are displays of including failure intensity (number of failures per unit time) versus failure number, actual and predicted cumulative number of failures versus time, and software reliability versus time.
4. After the models have been applied to a set of failure data, the next step is determining which model's results are the most appropriate to the data. CASRE provides a facility to help rank the models according to goodness of fit and four criteria described in [Abde86]—prequential likelihood, bias, bias trend, and model noise. The default ordering and weighting of these criteria is based on a study of how to rank software reliability model results [Niko95]. First, CASRE uses the goodness-of-fit tests to screen out those model results that don't fit at a specified significance level—those results are excluded from further consideration in the ranking. Next, CASRE compares the prequential likelihood values for the models—models with higher prequential likelihood values are ranked higher than those having lower values. For those models having the same prequential likelihood value, CASRE uses the extent to which the models are biased to break the tie—models having lower values of bias are ranked higher than models with higher bias values. For those models exhibiting the same amount of bias, CASRE uses the value of the bias-trend criterion to break the tie—ranking with respect to bias trend is the same as it is for bias. There is an additional criterion, "prediction noisiness," or "model noise," which is also available, but its meaning is not as well understood as that of the other three [Abde86]. The default use of this criterion is to break ties among models exhibiting identical values of bias trend—models having lower values of model noise are ranked higher than models having higher values. Model ranks are displayed in a tabular form—the overall rank for each model is displayed, as well as the ranks with respect to each individual criterion.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.

REFERENCES

[Abde86] A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood, "Evaluation of competing software reliability predictions," *IEEE Transactions on Software Engineering*, Vol. SE-12, pp. 950–967, September 1986.

[Jeli72] Jelinski, Z., and P. Moranda, "Software reliability research," in W. Freiberger (ed.), "Statistical Computer Performance Evaluation," Academic, 1972, pp. 465–484.

[Keen94] S. Keene, and G. F. Cole, "Reliability growth of fielded software," *Reliability Review*, Vol. 14, No. 1, March 1994.

[Lyu91] M. Lyu, "Measuring reliability of embedded software: An empirical study with JPL project data," *Proceedings of the International Conference on Probabilistic Safety Assessment and Management*, February 4–6, 1991.

[Lyu91a] M. R. Lyu, and A. P. Nikora, "A heuristic approach for software reliability prediction: The equally-weighted linear combination model," *Proceedings of the IEEE International Symposium of Software Reliability Engineering*, May 17–18, 1991.

[Lyu91b] M. R. Lyu, and A. P. Nikora, "Software reliability measurements through combination models: Approaches, results, and a CASE tool," *Proceedings of the 15th Annual International Computer Software and Applications Conference* (*COMPSAC91*), September 11–13, 1991.

[Lyu96] M. Lyu (ed.) "Handbook of Software Reliability Engineering," McGraw-Hill, 1996, pp. 493–504.

[Murp95] B. Murphy, and T. Gent, "Measuring System and Software Reliability Using an Automated Data Collection Process," Quality and Reliability Engineering International, CCC 0748-8017/95/050341-13pp., 1995.

[Musa87] J. D. Musa, A. Iannino, and K. Okumoto "Software Reliability: Measurement, Prediction, Application," McGraw-Hill, 1987.

[Niko95] A. Nikora, and M. R. Lyu, "An experiment in determining software reliability model applicability," *Proceedings of the 6th International Symposium on Software Reliability Engineering*. October, 24–27, 1995.

[Schn75] N. F. Schneidewind, "Analysis of error processes in computer software," *Proceedings of the International Conference on Reliable Software*, IEEE Computer Society, April 21–23, 1975, pp. 337–346.

[Schn92] N. F. Schneidewind, and T. W. Keller, "Applying reliability models to the space shuttle," *IEEE Software*, Vol. 9 pp. 28–33, July 1992.

[Schn93] N. F. Schneidewind, "Software reliability model with optimal selection of failure data," *IEEE Transactions on Software Engineering*, Vol. 19, November 1993, pp. 1095–1104.

[Schn97] N. F. Schneidewind, "Reliability modeling for safety-critical software," *IEEE Transactions on Reliability*, Vol. 46, March 1997, pp. 88–98.

[Shoo72] Shooman, M. L., "Probabilistic Models for software reliability prediction," in W. Freiberger (ed.), "Statistical Computer Performance Evaluation," Academic, 1972, pp. 485–502.

BIBLIOGRAPHY

Card, D. N., and R. L. Glass, "Measuring Software Design Quality," Prentice Hall, 1990.

Clapp, J. A., and S. F. Stanten, "A Guide to Total Software Quality Control, RL-TR-92-316," Rome Laboratory, 1992.

Friedman, M. A., P. Y. Tran, and P. L. Goddard, "Reliability Techniques for Combined Hardware and Software Systems," Hughes Aircraft, 1991.

McCall, J. A., W. Randell, J. Dunham, and L. Lauterback, "Software Reliability, Measurement, and Testing RL-TR-92-52", Rome Laboratory, 1992.

Shaw, M., "Prospects for an engineering discipline of software," *IEEE Software*, November 1990.

Downloaded from Digital Engineering Library @ McGraw-Hill (www.digitalengineeringlibrary.com)
Copyright © 2004 The McGraw-Hill Companies. All rights reserved.
Any use is subject to the Terms of Use as given at the website.