Software Reliability Engineering

Mladen A. Vouk

Mladen A. Vouk
Department of Computer Science, Box 8206
North Carolina State University, Raleigh, NC 27695

Tel: 919-515-7886, Fax: 919-515-7896 or 6497
e-mail: vouk@csc.ncsu.edu**, http://renoir.csc.ncsu.edu/Faculty/Vouk**

Summary & Purpose

Software-reliability engineering (SRE) stems from the needs of software users. The day-to-day operation of our society is increasingly more dependent on software-based systems and tolerance to *failures* of such systems is decreasing. Software engineering is not only expected to help deliver a software product of required functionality on time and within cost, it is also expected to help satisfy certain quality criteria. The most prominent one is *reliability*. SRE is the "applied science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction."

This tutorial first provides general information about the nature of SRE and how it relates to software process, including factors such as testing, cost and benefits. This is followed by an overview of the SRE terminology and modeling issues. Finally, SRE practice is addresses by discussing specifics of SRE activities required during different software life-cycle phases, including an example of how to generate SRE-based test-cases automatically.

Mladen A. Vouk

Mladen A. Vouk received B.Sc. and Ph.D. degrees from the King's College, University of London, U.K. He is a Professor of Computer Science at the N.C. State University, Raleigh, N.C., U.S.A. Dr. Vouk has extensive experience in both commercial software production and academic computing. He is the author, or co-author, of over 140 publications. His research and development interests include software engineering (software process and risk management, software testing and reliability), scientific computing (development of numerical and scientific software-based systems, parallel computing, scientific workflows), computer-based education (network-based education, distance learning, education workflows), and high-speed networks (end-user quality of service, forward error correction in high-speed networks, empirical evaluation of high-performance networking solutions). He teaches courses in software engineering, software testing and reliability, software process and risk management, and networking. He is closely associated with the N.C. State Multimedia and Networking Laboratory, and with the Computer Science Software Engineering and Computer-Based Education Laboratories. He is a member of the N.C. State Center for Advanced Computing and Communications.

He is a senior member of IEEE, a member of the IEEE Reliability, Communications and Computer Societies, a member of the IEEE TC on Software Engineering, and a member of ACM, ASQC, and Sigma Xi. Dr. Vouk is also a member of the IFIP Working Group 2.5 on Numerical Software. He is an associate editor of IEEE Transactions on Reliability. He is a member of the Editorial Board for the Journal of Computing and Information Technology, editor of the IEEE TCSE Software Reliability Engineering Newsletter, and a member of the Editorial Board for the Journal of Parallel and Distributed Computing Practices. He has been associated with the International Symposium on Software Reliability Engineering (ISSRE) since its inception. He was the General Chair of the 1992 symposium, Program Co-Chair for the 1995 event, Publications Chair for the 1996 event, and Tutorials Co-Chair for the 1997 event.

Table of Contents

1. Introduction

Software-reliability engineering (SRE) stems from the needs of software users. The day-to-day operation of our society is increasingly more dependent on software-based systems and tolerance to *failures* of such systems is decreasing. Software engineering is not only expected to help deliver a software product of required functionality on time and within cost, it is also expected to help satisfy certain quality criteria. The most prominent one is *reliability*. SRE is the "applied science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction" [Mus90, Lyu96, She97, Mus98].

2. About SRE

SRE is the focus of practical technology transfer efforts in many organizations with advanced software processes. For example, SRE is an accepted "best practice" for one of the major developers of telecommunications software (AT&T, Lucent). It is practiced in many other software development areas, including aerospace industry and network-based education [She97]. This increased interest in SRE is driven, at least in part, by the expectation that adoption of adequate SRE technology will increase the competitiveness of an organization or a project. There is mounting evidence that this is the case. The benefits include more precise satisfaction of customer needs, better resource and schedule control, and increased productivity.

Examples of organizations that are using, experimenting with, or researching SRE are Alcatel, AT&T, Lucent, Hewlett-Packard, Hitachi, IBM Corp., Jet Propulsion Laboratories, MITRE Corp., Motorola, NASA, NCR Corp., Nortel, Telcordia, U.S. Air Force, U.S. Navy, U.S. Army and Toshiba. Although direct economic information is usually difficult to obtain for proprietary reasons, studies show that the cost-benefit ratio of using SRE techniques can be six or more [Ehr93]. In one case, SRE has been credited with reducing the incidence of customer-reported problems, and maintenance costs, by a factor of 10. In addition, in the system-test interval the number of software-related problems was reduced by a factor of two, and in the product introduction interval by 30 percent. The same system showed no serious service outages within the first two years after its release, and a considerably increased customer satisfaction. Its sales were increased by a factor of 10, but only part of this is attributed to the increased quality [Abr92, Mus93].

It is estimated that routine application of SRE does not add more than several percent to the overall cost of a project. For example, a project involving 40 to 100 persons may require pre-project activities totaling about one to two person-weeks, definition of the operational profile(s) may require one to three person months, and routine collection and analysis of project failure and effort data may cost between one half to one person-day per week.

However, introduction of SRE into an organization will be a strong function of the (software process) maturity of that organization. Start-up costs may include deployment of an automated failure, fault and effort collection system, calibration of existing and development of organization-specific reliability models and tools, staff training, modification of the organizational culture, modifications in the employed software processes, etc. SRE introduction periods can range from six months to several years, again depending on the maturity of the organization and the available resources.

It is recommended that SRE be implemented incrementally. Starting point should be the activities needed to establish a baseline and learn about the product, about customer expectations, and about the constraints that the organizational business model imposes on its software production [Pot97]. The initial effort includes collection of basic failure data, monitoring of reliability growth during system tests, field trials and software operation, and the initial formulation of operational profiles. This should be followed by the development of detailed operational profiles, detailed classification of system failures and faults, and development of business-based reliability objectives. More advanced stages involve continuous tracking of customer satisfaction, trade-off studies, quantitative evaluation of software process capabilities with respect to reliability, and proactive process control.

3. Basic Terms

Software-reliability engineering is the quantitative study of the operational behavior of software-based systems with respect to user requirements. It includes

(1) Software reliability measurement (assessment) and estimation (prediction);
(2) Effects of product and development process metrics and factors (activities) on operational software behavior;
(3) Application of this knowledge in specifying and guiding software development, testing, acquisition, use, and maintenance.

Reliability is the probability that a system, or a system component, will deliver its intended functionality and quality for a specified period of *"time"*, and under specified conditions, given that the system was functioning properly at the start of this *"time"* period. For example, this may be the probability that a real-time system will give specified functional and timing performance for the duration of a ten hour mission when used in the way and for the purpose intended. Since, software reliability will depend on how software is used, software usage information is an important part of reliability evaluation. This includes information on the environment in which software is used, as well as the information on the actual frequency of usage of different functions (or operations, or features) that the system offers. The usage information is quantified through *operational profiles*.

"Time" is execution exposure that software receives through usage. Experience indicates that the best metric is the actual central processing unit (CPU) execution-time. However, it is possible to reformulate measurements, and *reliability models*, in terms of other exposure metrics, such as calendar-time, clock-time, number of executed test cases (or runs), fraction of planned test cases executed, inservice-time, customer transactions, or structural coverage. In considering which "time" to use, it is necessary to weigh factors such as availability of data for computation of a particular metric, error-sensitivity of the metric, availability of appropriate reliability models, etc. An argument in favor of using CPU time, or clock-time, instead of, for example, structural software coverage, is that often engineers have a better physical grasp of time, and, in order to combine hardware and software reliabilities, the time approach may be essential. On the other hand, it may make more sense to use "printed pages" as the exposure metric when dealing with reliability of printers.

When a system in operation does not deliver its intended functionality and quality, it is said to fail. A **failure** is an observed departure of the external result of software operation from software requirements or user expectations [IEE88a, IEE88b, IEE90]. Failures can be caused by hardware or software faults (defects), or by how-to-use errors.

A **fault** (or defect, or bug) is a defective, missing, or extra instruction, or a set of related instructions, that is the cause of one or more actual or potential failures. Inherent faults are the faults that are associated with a software product as originally written, or modified. Faults that are introduced through fault correction, or design changes, form a separate class of modification faults. An associated measure is **fault density** — for example, the number of faults per thousand lines of executable source code. Faults are the results of (human) **errors**, or mistakes. For example, an error in writing a programming language branching statement, such as an if-statement condition, will result in a physical defect in the code, or fault, that will on execution of that statement transfer control to wrong branch. If, on execution, such a program does not produce the desired results, for example display a particular picture, it is said to fail and a failure has been observed.

How-to-use errors. Failures can be caused by software faults, functional lacks in software, or user errors (for example, lacks in user's knowledge). It is important to understand that failures and how-to-use errors, and their frequency, tend to relate very strongly to customer satisfaction and perception of the product quality. On the other hand, faults are more developer oriented, since they tend to be translated into the amount of effort that may be needed to repair and maintain the system.

Severity of a failure or fault is the impact it has on the operation of a software-based system. Severity is usually closely related to the threat the problem poses in functional (service), economic (cost) terms, or in the case of critical failures, to human life. An example of a service impact classification is: critical, major and minor failure. Severity of failures (or faults) is sometimes used to subset the operational failure data, and thus make decisions regarding failures of a particular severity, or to weight the data used in reliability and availability calculations.

Operational profile is a set of relative frequencies (or probabilities) of occurrence of disjoint software operations during its operational use. A detailed discussion of operational profile issues can be found in [Mus87, Mus93, Mus98]. A software-based system may have one or more operational profiles**.** Operational profiles are used to select **test cases** and direct development, testing and maintenance efforts towards the most frequently used or most risky components. Construction of an operational profile is preceded by definition of a **customer** profile, a **user** profile, a system **mode** profile, and a **functional** profile. The usual participants in this iterative process are system engineers, high-level designers, test planners, product planners, and marketing. The process starts during the requirements phase and continues until the system testing starts. **Profiles** are constructed by creating detailed hierarchical lists of customers, users, modes, functions and operations that the software needs to provide under each set of conditions. For each item it is necessary to estimate the probability of its occurrence (and possibly **risk** information) and thus provide a quantitative description of the profile. If usage is available as a rate (e.g., transactions per hour) it needs to be converted into probability. In discussing profiles, it is often helpful to use tables and graphs and annotate them with usage and criticality information.

4. Metrics and Models

A significant set of SRE activities are concerned with measurement and prediction of software reliability and availability. This includes, modeling of software failure behavior, and modeling of the process that develops and removes faults. A number of metrics and models are available for that purpose [Mus98, Lyu96, Mus87, IEE88a, IEE88b, Mal91, Xie91, AIA93]. This section examines the basic ideas.

4.1 Reliability

We distinguish two situations. In one situation, detected problems are further pursued and fault identification and correction takes place, for example, during software development, system and field testing, and active field maintenance. In the other situation, no fault removal takes place, for example, between successive releases of a product. In the first case we would expect the product to improve over time, and we talk about **reliability growth**.

The quality of software, and in particular its reliability, can be measured in a number of ways. A metric that is commonly used to describe software reliability is *failure intensity*. **Failure intensity** is defined as the number of failures experienced per unit "time" period. Sometimes the term **failure rate** is used instead. An interesting associated

measure is the mean time to failure. Often **mean time to failure** is well approximated by the inverse of the failure intensity or failure rate. Failure intensity can be computed for all experienced failures, for all unique failures, or for some specified category of failures of a given type or severity. Failure intensity is a good measure for reflecting the user perspective of software quality. When reliability growth is being experienced, failure intensity will decrease over time.

When there is no repair, it may be possible to describe the reliability of a software-based system using **constant** failure intensity, λ, and a very simple exponential relationship:

$$R(t) \sim e^{-\lambda\tau} \quad (1)$$

where R(t) is the reliability of the system, and "τ" is the duration of the mission. For example, suppose that the system is used under representative and unchanging conditions, and the faults causing any reported failures are not being removed. Let the number of failures observed over 10,000 hours of operation be 7. Then, failure intensity is about $\hat{\lambda} = 7/10000 = 0.0007$ failures per hour, and the corresponding mean time to failure is about $1/\lambda = 1428$ hours. From equation (1), and given that the system operates correctly at time $t = 0$ hours, the probability that the system will **not** fail during a 10 hour mission is about $R(10) = e^{-0.0007*10} = 0.993$.

Where software **reliability growth** is present, failure intensity, $\lambda(\tau)$, becomes a decreasing function of time τ during which software is exposed to testing and usage under **representative** (operational) conditions. There is a large number of software reliability models that address this situation [Lyu96], but before any modeling is undertaken, it is a good idea to confirm the presence of the growth using **trend** tests [Mus87, Kan93b]. All models have some advantages and some disadvantages. It is extremely important that an appropriate model be chosen on a case by case basis [Mus87, Bro92, Lyu96].

Two typical models are the "basic execution time" (BET) model [Goe79, Mus87] and the Logarithmic-Poisson execution time (LPET) model [Mus84, Mus87]. Both models assume that the testing uses operational profiles, and that every detected failure is immediately and perfectly repaired[1]. The **BET** failure intensity $\lambda(\tau)$ with exposure time τ is:

$$\lambda(\tau) = \lambda_0 e^{-\frac{\lambda_0}{\nu_0}\tau} \quad (2)$$

where λ_0 is the initial intensity and ν_0 is the total expected number of failures (faults). It is interesting to note that the model becomes linear if we express intensity as a function of cumulative failures

$$\lambda(\tau) = \lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right) \quad (3)$$

where $\mu(\tau)$ is the mean number of failures experienced by time τ, or the **mean value function**, i.e.

$$\mu(\tau) = \nu_0\left(1 - e^{-\frac{\lambda_0}{\nu_0}\tau}\right) \quad (4)$$

On the other hand, the **LPET** failure intensity $\lambda(\tau)$ with exposure time τ is:

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \quad (5)$$

where λ_0 is the initial intensity and θ is called the failure intensity decay parameter since

$$\lambda(\tau) = \lambda(\mu) = \lambda_0 e^{-\theta\mu} \quad (6)$$

and $\mu(\tau)$ is the mean number of failures experienced by time τ, i.e.,

$$\mu(\tau) = \frac{1}{\theta}\ln(\lambda_0 \theta \tau + 1)) \quad (7)$$

The BET model represents a class of "**finite-failure**" models for which the mean value function tends towards a level asymptote as exposure time grows, while the LPET model is a representative of a class of models called "**infinite-failure**" models since it allows an unlimited number of failures. Of course, both classes of models can be, and are being, used to describe software fault removal processes that may involve only a finite number of actual faults [Jon93].

Given failure intensity data, it is possible to estimate model parameters. Estimation can be made in many ways. Two common methods are **maximum likelihood** and **least squares** [Mus87]. It is very important to understand that there are two distinct ways of using a model. One is to provide a **description** of historical (already available) data. The other is to **predict** future reliability measures and events during actual testing or operation, such as "when will the intensity reach a target value", or "when can I stop testing". Predictions are more interesting from a practical standpoint, but also the more dangerous. Brocklehurst and Littlewood note that no single model can be universally recommended, and accuracy of reliability measures produced by a model can vary greatly. However, there are advanced statistical techniques, such as **u-plots** and **prequential likelihood ratio**, that can alleviate the accuracy problem to some extent [Bro92. Lyu96].

[1]There are some other assumptions that have to be satisfied (see Mus87). Also, there are model variants that operate with different assumptions, such as delayed and less than perfect fault repair.

Once a model has been selected and its parameters estimated, it is possible to compute quantities such as the total number of faults in the code, future failure intensity and, given a target intensity, how much longer the testing needs to go on. For instance, suppose that it was determined that the BET model is appropriate. Then it follows from equation (2) - (4) that the number of additional failures, $\Delta\mu$, that must be experienced to achieve failure intensity objective λ_F is

$$\Delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F), \quad (8)$$

where λ_P is the present failure intensity. Similarly, the additional execution time, $\Delta\tau$, required to reach the failure intensity objective is

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln\frac{\lambda_P}{\lambda_F} \quad (9)$$

For example, assume that it is estimated that there are a total of ν_0=120 faults in the code, that λ_0= 15 failures per CPU hour, that λ_P = 2.5 failures per CPU hour, and the objective is to achieve 0.0005 failures per CPU hour. Then, $\Delta\mu = \frac{120}{15}(2.55 - 0.0005) \sim 21$ failures, and $\Delta\tau = \frac{120}{15} \ln\frac{2.55}{0.0005} \sim 68.3$ CPU hours. If it is known what effort expenditure is required to detect a failure, identify and correct the corresponding fault, and how much cost is associated with exposure time, it is possible to construct economic models that relate the testing not only to the resultant quality, but also to the expended effort (cost) [Mus87, Ehr93, Yam93].

The estimates given in the above example are known as **point estimates** since they involve only the "most likely" or the "best" value. However, in practice, it is extremely important to compute confidence bounds for any estimated parameters and derived quantities in order to see how much one can rely on the obtained figures [Mus87]. This involves computation of probable errors (variances) for both the model parameters and the derived quantities. Instead of presenting the projections as single values we need to present them as an appropriate **interval** (e.g., 70%, 90% or 95% confidence interval). For example, instead of saying that 21 failures are expected to occur before we reach the target intensity, we might use the 90% interval, say from 17 to 25 failures, or [17, 25]. It is essential that a person selecting models and making reliability predictions is appropriately trained in both software (reliability) engineering and statistics.

Since a large fraction of the variability in the estimates usually derives from the variability in the collected data, accurate and comprehensive data collection is of ultimate importance. For example, data collection should include the times of successive failures (alternatively intervals between failures may be collected, or the number of failures experienced during an interval of testing — grouped data — may be recorded), information about each corrected fault, information about the parts of the code and product modules affected by the changes, information about the expended effort, etc.

It is recommended that both the data collection and the model estimation be automated and tool-based. Examples of reliability oriented data-sets and tools can be found on the CD-ROM that comes with the Handbook of Software Reliability Engineering [Lyu96]. Examples of tools that can aid in software reliability estimation are SMERFS [Far88, Lyu96] and RelTools [Mus90] on Unix , CASRE on DOS and Windows [Lyu92, Lyu96], and SoRel on Macintosh computers [Kan93a. Lyu96]. Example of a tool that can help in test-case development, and that we discuss further later in this tutorial, is PairTest [Lei98].

Figure 1 illustrates maximum likelihood fits for BET and LPET models to a well known system test data set called T1 [Mus87]. The plot is of the natural logarithm of failure intensity vs. execution time. It is also quite common to plot failure intensity against cumulative failures to see if the relationship given in equation (2) holds. While graphs can be used to screen the data for trends, statistical tests must be used to actually select a model [Mus87, Bro92]. In this case the tests show that the LPET model fits somewhat better than the BET model. However, in a different project the BET, or some other model, may be better than the LPET model. Figure 2 shows the cumulative failure distribution obtained from the data and the models.

SRE models tend to assume exposure (testing) based on an operational profile. Since this assumption is usually violated during early software testing phases (for example, during unit-testing and integration-testing), assessment and control of software quality growth during non-operational testing stages is difficult and open to interpretation. In an organization that constructs its final deliverable software out of a number of components that evolve in parallel, an added problem can be the variability of the quality across these components.

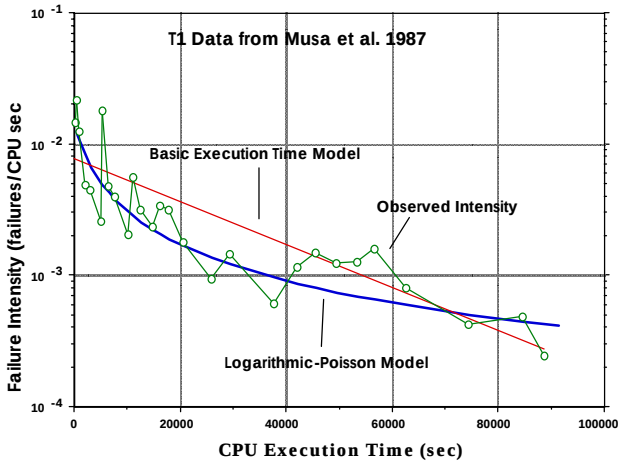


Figure 1. Empirical and modeled failure intensity.

Another confounding factor can be the (necessary) discontinuities that different testing strategies introduce within one testing phase, or between adjacent testing phases. For instance, unit-testing concentrates on the functionality and coverage of the structures within a software unit, integration-testing concentrates on the coverage of the interfaces, functions and links that involve two or more software units, etc. It is not unusual to observe an apparent failure-intensity decay (reliability growth) during one of the phases, followed by an upsurge in the failure-intensity in the next phase (due to different types of failures). This oscillatory effect can make reliability growth modeling difficult, although several different approaches for handling this problem have been suggested [e.g., Mus87, Lyu92. Lyu96] .

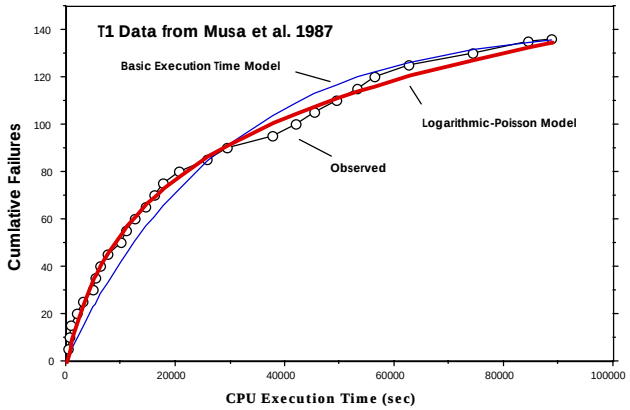


Figure 2. Observed and modeled cumulative failures.

A large class of models that can be useful in the context of early testing phases, and non-operational testing in general, are the so called "**S-shaped**" models that describe failure intensity that has a mode or a peak [Yam83, Ohb84, Mus87, Yam93]. These models derive their name from the S-like shape of their cumulative failure distributions. Figures 3 and 4 illustrate use of a Weibull-type model [Mus87] during unit and integration testing phases of a telecommunications software product [Vou93, Lyu96].

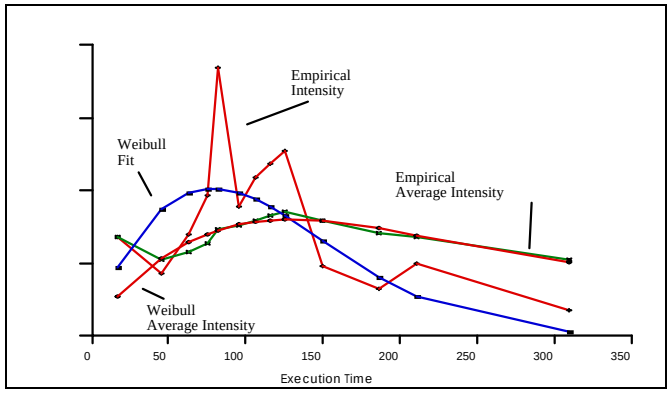


Figure 3 Empirical and modeled intensity profiles obtained during an early testing phase. Exposure is the cumulative test case execution time "t". Average intensity at time "t" is the total number of failures experienced by "t" divided by the total execution time.

The need to recognize software problems early, so that appropriate corrections (process feedback) can be undertaken within a single software release frame, is obvious. How to achieve this is less clear. In general, it is necessary to link the symptoms observed during the early testing phases with the effects observed in the later phases, such, as identification of components that may be problem-prone in the early operational phase. Several authors have published models that attempt to relate some early software metrics, such as, the size of the code, Halstead length, or cyclomatic number, to the failure proneness of a program [Kho90, Mun92, Bri93]. A more process-oriented approach is discussed in [Vou93, Lyu96]. Highly correlated nature of the early software verification and testing events may require the use of a more sophisticated, time-series, approach [Sin92].

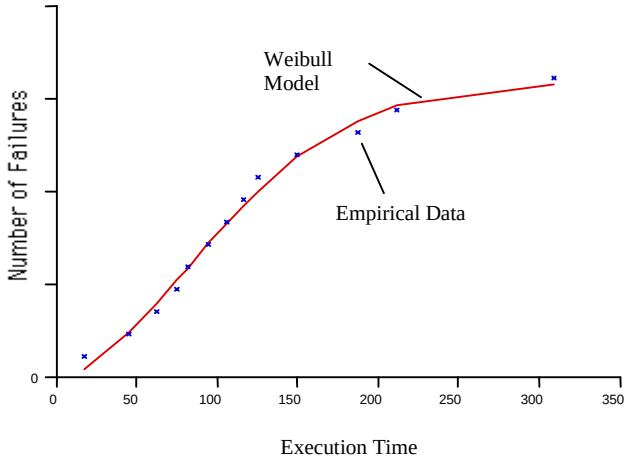


Figure 4 Empirical and modeled failures obtained during an early testing phase.

4.2 Availability

Another important practical measure for software quality is *availability*. For example, the Bellcore[2] unavailability target for telecommunications network elements is about 3 minutes of downtime per year. **Availability** is the probability that a system, or a system component, will be available to start a mission at a specified "time" [Fra88]. **Unavailability** is the opposite, the probability that a system or a system component will **not** be available to start a mission at a specified "time". The concept of (un)availability is closely connected to the notion of repairable failures.

Recovery from failures can be expressed through recovery or **repair rate**, ρ, that is the number of repaired failures per unit time. For example, software failures may

[2]Bellcore is an organization that acts as a software quality "watchdog" from within the U.S. telecommunications community.

result in a computer system outages that, on the average, last 10 minutes each before the systems is again available to its users. The estimated repair, or recovery, rate for the system is then 1/10 failures per minute. System availability can be expressed in several ways. For example, **instantaneous availability** is the probability that the system will be available at any random time t during its life. **Average availability** is the proportion of time, in a specified interval [0,T] that the system is available for use [San63].

We can estimate average software availability in the period [0, T] as:

$$\hat{A}_c(T) = \frac{\text{Total time software operated correctly in the given period}}{T} \quad (10)$$

Associated with average availability are average failure ($\hat{\lambda}_c(T)$) and recovery rates ($\hat{\rho}_c(T)$) estimates

$$\hat{\lambda}_c(T) = \frac{\text{Total Number of Failures in Period T}}{\text{Total Time System was Operational During Period T}} \quad (11)$$

$$\hat{\rho}_c(T) = \frac{\text{Total Number of Failures in Period T}}{\text{Total Time System was under Repair or Recovery During Period T}} \quad (12)$$

If the period T is long enough, the average availability approaches **steady state availability**, A_{ss} which, given some simplifying assumptions, can be described by the following relationship [Tri82, Sho83]):

$$A_{ss} = \frac{\rho}{\lambda+\rho} \quad (13)$$

We see that two measures which directly influence the availability of a system are its failure rate (or **outage** rate as failure rate is sometimes called) and its field repair rate (or software recovery rate). Figure 5 shows failure and recovery rates observed during operational use of a telecommunications product [Cra92]. Apart from the censored[3] "raw" data two other representations are shown. In one, the data are smoothed using an 11-point symmetrical moving average. In the other, we show cumulative average of the data. Note that immediately after the product release date, there is considerable variation in the failure rate. This is the **transient** region. Later the failure rate reduces and stabilizes. In a system which improves with field usage we would expect a decreasing function for failure rate with **inservice time**[4] (implying fault or problem reduction and reliability growth).

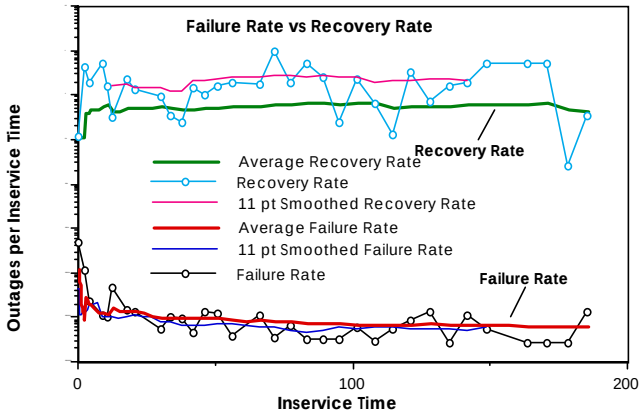


Figure 5. Field recovery and failure rates for a telecommunications product.

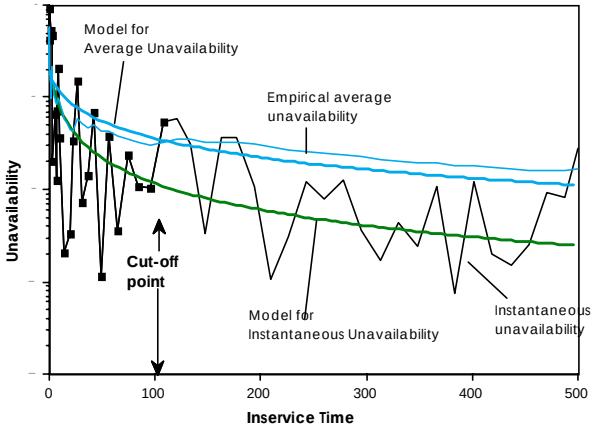


Figure 6. Unavailability fitting using LPET and constant repair rate with data **up to** "cut-off point" only.

Field failure rate is usually connected to both the operational usage profile and the process of problem resolution and correction. Recovery rate depends on the operational usage profile, the type of problem encountered, and the field response to that problem (i.e., the duration of outages in this case). It is not unusual for a recovery rate to be 3 to 4 orders of magnitude larger than the failure rate.

In practice, reliability and availability models would be used to predict future unavailability of a system. Of course, only the data up to the point from which the prediction is being made would be available. The prediction would differ from the true value depending on how well the model describes the system. We illustrate this in Figure 6. It shows the empirical unavailability data and fits for two simple models. The fits are based on the average recovery rate observed at the "cut-off point", and the LPET failure fit to the points from the beginning of the release's operational phase up

[3] Zero valued data points are not shown in order to allow the use of logarithmic scale on the ordinate.

[4] Total time the software-based system was in service, that is, either operating correctly or going through a repair or recovery episodes, at all sites that have the software installed.

to the "cut-off point". The figure shows that, in this case, both models appear to predict future system behavior well. The models are described in [Cra92]. Other models are available, e.g., [Lap91].

The point to note is that a relatively simple model can have quite reasonable predictive properties for a system that has known history (through multiple releases) and is maintained in a stable environment.

5. Practice

5.1 Verification and Validation

It is not feasible to practice SRE without sound and solid *software verification and validation* plan, process and activities throughout software life-cycle. An example of such a plan can be found in the IEEE software engineering standards [IEE86]. SRE implies use of modern fault-avoidance, fault-identification, fault-elimination, and fault-tolerance technology. SRE extends that technology through quantification and matching with the business model. This includes construction and quantification of software usage profiles, and specification of a balance between software reliability and other constraints. SRE practices require collection and analysis of data on software product and process quality, estimation and tracking of reliability, guidance of software development processes in terms of resources and "when-to-stop" testing information, and monitoring of software field reliability. SRE tracking and analyses are used to improve organizational software development and maintenance process and maximize customer satisfaction. The following paragraphs provide an overview of the principal SRE activities during a typical [Pre97] software life-cycle.

The IEEE verification and validation (V&V) standard suggests that following V&V tasks be conducted during a **software requirements specification and analysis** phases: i) software requirements traceability analysis, ii) software requirements evaluation, iii) software requirements interface analysis, iv) system test plan generation, and v) software acceptance test plan generation. SRE augments these activities by requiring that the developers, in conjunction with the customer (users) need to a) explicitly define and categorize software **failure modes**, b) determine **reliability needs** of the customer and analyze the economic trade-offs (schedule vs. quality vs. cost), c) determine software usage profile(s), and d) set **reliability targets** for the product.

Identification and classification of software **failure modes** and their severity will depend on the application, customers, and maintenance needs. For example, U.S. Federal Communications Commission requires service disruptions, such as loss of telephone service, that exceed 30 minutes and affect more than 50,000 customer to be reported to FCC within 30 minutes of its occurrence [FCC92]. In this context, a telephone switch failure may be classified, in descending order of severity, as 1) FCC-reportable, 2) complete system outage of less than 30 minutes or affecting less than 50,000 customer lines, 3) loss of one or more principal functionalities or services, 4) loss of functionality that allows use of back-up or workaround options, 5) minor inconvenience or inefficiency. In order to diagnose the **root causes** of a failures it is important to gather failure data. A failure that caused a complete system outage, may be further sub-classified by its identified or hypothesized cause into hardware-caused, software-caused, procedural (e.g., the system operator made a mistake and accidentally shut the system down), or unknown. Within each sub-class it is possible to have additional categories. For example, if system availability (or repair time) is important, it may be advantageous to classify failures by their duration (e.g., less than 2 minutes, 2 to less than 5 minutes, 5 to less than 10 minutes, 10 to less than 30 minutes, 30 or more minutes).

Reliability target is established by considering the needs of customers as well as the limitations of the software engineering technology, capabilities of the developers, and other constraints such as the oerganizational **business model** and development costs and schedules. Usually, separate objectives are set for reach software failure category. For example, Bellcore has established generic requirements for performance evaluation of telecommunications systems [Bel90]. The Bellcore target for a network switching element (e.g., a telephone exchange) is about 3 minutes of downtime per year and a complete outage failure is recognized if the services is down for over 30 seconds. Of course, this is not just a reliability requirement but also an availability requirement. In fact, it specifically average unavailability for the system of about 3/(60*24*365).= .00000571. To compute the reliability target it is necessary to also establish a target value for system repair. For instance, under a simplifying assumption that the system has actually stabilized in its steady state, we can use the relationship (13) to set a possible reliability target. Let the average system repair rate be ρ=0.3 failures per minute. Then substituting required availability 1-0.00000571 = .99999429 and known repair rate into (13), and solving for the average system failure rate we find that it should not exceed .00000171 failures per min. Practice shows that this is a reasonable and achievable target for telecommunication software.

The IEEE standard also suggests that the following tasks be conducted during a software **design, coding, unit testing and integration testing** phases: i) design and code traceability analyses, ii) evaluations of software designs, code and documentation, iii) software interface analyses, iv) generation of test plans for software components and for their integration, and v) design, generation and execution of test cases. SRE augments these activities by requiring software developers to a) finalize functional and define **operational profiles**, b) evaluate reliability of software components that are **"re-used"**, c) explicitly **allocate reliability** among software components and engineer the product to meet reliability objectives, d) use **resource and schedule** models to guide development workload according to functional profiles, and e) track and manage **fault introduction and removal** process.

An important activity in the design phase is **allocation of reliability** objectives to sub-systems and components in such as way that the total system reliability objective is achieved. The allocation approach should be iterative and it should include consideration of alternative solutions. The "balance" one is looking for is between the overall system reliability, and the development schedule and effort (cost). Available options include inclusion of good exception handling capabilities and use of different fault-tolerance techniques [Pha92, Lyu94] combined with systems and software risk analysis and management [Boe89].

Use of **inspections** and **reviews** is highly recommended in all phases of the software development process [IEE86]. They provide a means of **tracking and managing** faults during the stages where the software is not in executable form, as well as in the stages where it is. A rule-of-thumb metric is the number of major faults per person-hour spent on preparation and conduct of inspections. If this metric is in the range 3 to 7, the inspection process as well as the software process is probably under control, otherwise some corrective action is needed. More details can be found in [IEE86, Chr90]. Once executable software is available tracking and management can be supplemented using reliability models and reliability control charts.

Evaluation of the reliability of **legacy code** and of any **"acquired" or "re-used" code**, using operational profiles appropriate for the current application, is also recommended to ensure that the reliability of the "inherited" code is still acceptable. A special form of control charts may be used to monitor progress and decide on whether to accept or reject the components [Mus87].

5.2. Operational Profile

A crucial activity is definition of **operational profiles** and associated test cases. The process involves definition of customer, user and system-mode profiles, followed by the definition of functional and operational profile(s). For example, a customer is a person, a group, or an institution that acquires the system. The following table illustrates a hypothetical customer profile for telephone switch software. We show two categories of customers and the associated probability that the customer will use the product. The probability information could come from actual measurements, or it could be gleaned from sales data.

Hypothetical Telephone Switch

Customer group	**Probability**
Local Carrier	0.7
Inter-City Carrier	0.3

The next step is to identify the users. A user is a person, a group, or an institution that employs the system. Users are identified within each customer category. For example:

User group	**Local (0.7)**		**Inter-City (0.3)**		**Total**
	within	total	within	total	
Households	0.6	0.42	0.2	0.06	0.48
Businesses	0.3	0.21	0.6	0.18	0.39
Emergency Services	0.05	0.035	0.001	.0003	.0353
Other	0.05	0.035	0.199	.0597	.0947

For instance, the above table shows that within the local carrier category 60% of the users are households, 30% businesses, 5% emergency services, and 5% other users. The contribution that local households make to the total traffic is 42% (0.7*0.6 = 0.42), the contribution that inter-city household calls make to the total traffic is 6%, and the household user class as a whole accounts for 48% of the total switch usage.

The system can be used in several modes. A system mode is a set of functions or operations that are grouped for convenience in analyzing execution behavior [Mus90]. System can switch among modes, and two or more modes can be active at any one time. The procedure is to determine the operational profile for each system mode. Modes can be defined on the basis of user groups, environment (e.g., overload vs. normal, high-noise vs. low-noise), criticality, user experience, platform, how user employs system operations to accomplish system functions, etc. For example, consider the following table. Assume that 99% of inter-city and 90% of local household traffic is voice, while only 70% of business traffic is voice for both customer categories. Furthermore assume that system administration accounts for 30% and maintenance for 70% of the "Other" user category, while the rest of the traffic is DATA). Then

Mode		**Probability**
Voice (personal)	0.4374	*0.42*0.9+0.06 *0.99*
Voice (business)	0.273	*0.39*0.7*
Data	0.1596	*0.42*0.1 + 0.06*0.01 + 0.3*0.39*
Emergency	0.0353	0.0353
System Admin.	0.02841	*0.0947*0.3*
Maintenance	0.06629	*0.0947*0.7*

To obtain functional profile it is now necessary to break each system mode into user-oriented functions needed for its operations and associated probabilities (e.g., features plus the environment). A function may represent one or more tasks, operations, parameters, or environmental variables (e.g., platform, operating system). The list of functions should be kept relatively short (e.g., from about 50 to several hundred). It should be noted that the functions will tend to evolve during system development and the profiling process is an iterative one.

The final two steps are the definition of an operational profile through explicit listing of operations and generation of test cases. The operations are the ones that are tested. Their profile will determine verification and validation resources, test cases and the order of their execution. The operations need to be associated with actual software commands and input states. These commands and *input states* are then sampled in accordance with the associated probabilities to generate test cases. Particular attention should be paid to generation of test cases that address critical and special issues.

Operational profiles should be updated on a regular basis since they can change over time, and the number of operations that are being tested should be limited. Probability may not be the only criterion for choosing the profile elements. Cost of failure (severity, importance) of the operations plays a role. In fact, separate profiles should be generated for each category of criticality (typically four separated by at least an order of magnitude in effects) [Mus90].

5.3 Testing

Input space for a program is the set of discrete **input states** that can occur during the operation of the program. The number of dimensions of the input space is equal to the sum of the dimensions of the input variables. An input variable is any data item that exists external to a program is used by the program, while an output variable is any data item that exists external to the program and is set by the program [Mus87]. Note that in addition to the usual program parameters, externally initiated interrupts are also considered as input variables. Intermediate data items are neither input nor output variables.

5.3.1 Generation of Test Cases

In principle, one would use the operational profile of a product to identify the most important **scenarios** for the system and the corresponding operations and associated input states and thus develop test cases. Operational profile based testing is quite well behaved, and when executed correctly, allows dynamic evaluation of software reliability growth based on "classical" reliability growth metrics and models and use of these metrics and models to guide the process [Mus87, Mus98].

Unfortunately, in reality things may not be so straightforward. The challenges of modern "market-driven" software development practices, such as the use of various types of incentives to influence workers to reduce time to market and overall development cost, seem to favor a resource constrained approach which is different from the "traditional" software engineering approaches [Pot97]. In this case, the testing process is often analogous to a "sampling without replacement" of a finite (and sometimes very limited) number of **pre-determined** input states, data and other structures, functions, and environments [Riv98a, Riv98b]. The principal motive is to verify required product functions (operations) to an **acceptable** level, but at the same time minimize the re-execution of already tested functions (operations). This is different from strategies, that advocate testing of product functions according to the relative frequencies of their usage in the field, or according to their operational profile discussed in the previous subsection [Mus98]. These testing strategies tend to allow for much more re-execution of previously tested functions and the process is closer to a "sampling with (some) replacement" of a specified set of functions.

In an "ideal" situation the test-suite may consist only of test-cases that are capable of detecting all faults, and defect removal is instantaneous and perfect. In "best" cases sampling without replacement, the one that usually operates in resource constrained situations, requires less test-steps to reach a desired level of "defects remaining" than methods that re-use test-steps (or cases) or re-cover already tested constructs, such as those based on operational profiles. However, in practice, it may turn out that test-cases constructed using operational profiles may be more efficient and more comprehensive than those constructed using some non-ideal coverage-based strategy. If not executed properly, testing based on sampling without replacement will yield poorer results.

For instance, when there are deviations from the "ideal" sampling without replacement, as is usually the case in practice a number of defects may remain uncovered by the end of the testing phase. Figure 7 illustrates the differences.

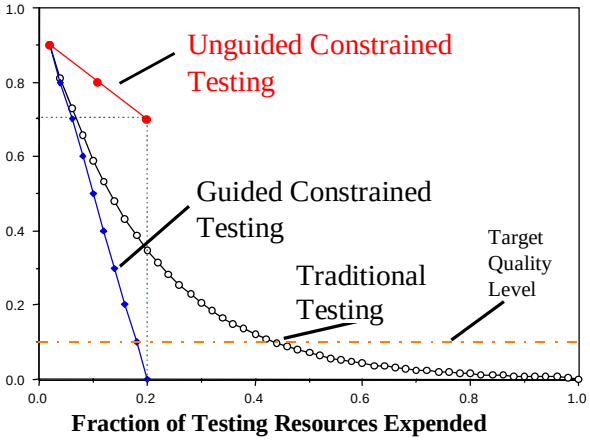


Figure 7. Fraction of shipped defects (y-axis) for two "ideal" testing strategies based on sampling with and without replacement, and a "non-ideal" testing under schedule and resource constraints.

"Unguided" constrained testing illustrates a failed attempt to cut the resources to about 20% of the resources that might be needed for complete operational-profile based test of a product. The selection of the test cases suite was inadequate, and although the testing is completed within the required resource constraints, it only detects a small fraction of the latent faults (defects). In the field, this product will be a constant emitter of problems and its maintenance will probably cost many times the resources "saved" during the testing phases. **"Traditional" testing** illustrates ideal testing based on operational profile that detects all faults present in the software but takes up more resources. Finally, **"guided"**

constrained testing illustrates an ideal situation where every test case reveals a fault and no resources or time is wasted.

A good way to develop "guided" test-cases is to start with a test-suite based on the operational profile and trim the test cases in a manner that preserves coverage of important parameters and coverage measures. One such approach is discussed by Musa in his book [Mus98]. Another one is to use pair-wise test-case generation systems [Coh94, Coh96, Coh97, Lei98]. Of course, there are many other possible approaches, and many of the associated issues are still research topics.

5.3.2 Pair-wise Testing

Pair-wise testing is a specification-based testing strategy which requires, in principle, that every combination of valid values of any two input parameters of a system be covered by at least one test case. Empirical results show that pair-wise testing is practical and effective for various types of software systems [Coh94, Coh96, Coh97]. According to Lei and Tai the Pair-wise testing steps are as follows [Lei98]:

a) "For each input parameter, specify a number of valid input values. If a parameter has a large number of valid values, choose representative and boundary values. The first value of each parameter must be a representative value of the parameter".

b) "Specify a number of relations for input parameters, where a relation is a set of two or more related input parameters. An input parameter may be in two or more relations. If an input parameter does not appear in any relation, it is called a non-interacting parameter. For each relation, constraints can be provided to specify prohibited combinations of values of some parameters in the relation. Each constraint is defined as a set of values for distinct parameters".

c) "Generate a test set for the system to satisfy the following requirements: (i) For each relation, every allowed combination of values of any two parameters in the relation is covered by at least one test, (ii) For each non-interacting parameter, every value of the parameter is covered by at least one test, (iii) Each test does not satisfy any constraint for any relation, and (iv) The first test contains the first value of each parameter".

For example, a system was found to have parameters A, B, and C as the most important parameters according to the operational profile studies (it could also be that it only has these three parameters at the level of abstraction at which the testing is being conducted). Let the most important (e.g., most frequent, or highest risk, or all, etc) values be

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	

Then, we will need 3x3x2 = 18 test cases if all three parameters are related (interacting) and we wish to cover all combinations of the parameters. On the other hand, pair-wise testing strategy requires only nine (9) tests to cover all PAIRS of combinations at least once.

A	B	C
A1	B1	C1
A1	B2	C2
A1	B3	C1
A2	B1	C2
A2	B2	C1
A2	B3	C2
A3	B1	C1
A3	B2	C2
A3	B3	C1

One can, of course, add constraints and further reduce the number of test cases. For example, if combination (A3, B3) is forbidden, the last test case could be deleted without affecting the coverage of (A3,C1) and (B3,C1), since they are covered by (A3,B1,C1) and (A1,B3,C1), respectively. As the number of parameters grows, the number of test cases required by pair-wise testing strategy grows linearly with the number of parameters rather than exponentially as it does with strategies which execute all combinations of the parameters (see the example in Slides 45 and 46 in the Appendix.

Pair-wise testing can be used for different levels of specification-based testing, including module testing, integration testing, and system testing. It is also useful for specification-based regression testing (see example in the Appendix). Different levels of testing for a system have different sets of input parameters. The number of tests generated for pair-wise testing of a program unit depends upon the number of input parameters, the number of values chosen for each input parameter, the number of relations, and the number of parameters in each relation.

PairTest is a software tool that generates a test set satisfying the pairwise testing strategy for a system [Lei98]. The major features of PairTest include the following:

- *"PairTest supports the generation of pairwise test sets for systems with or without existing test sets and for systems modified due to changes of input parameters and/or values.*
- *PairTest provides information for planning the effort of testing and the order of applying test cases.*
- *PairTest provide a graphical user interface (GUI) to make the tool easy to use.*
- *PairTest is written in Java and thus can run on different platforms."*

The PairTest tool was developed by Dr. K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University [Lei98, http://renoir.csc.ncsu.edu/Tools/]. Another such tool is AETG [Coh97]. PairTest uses a somewhat different test-case generation algorithm than does AETG.

5.4 Process

The importance of continuous software reliability evaluation is in establishing quality conditions which can be used for software process control. In the **system** and **field**

testing phases standard activities include: i) execution of system and field acceptance tests, ii) checkout of the installation configurations, and, iii) validation of software functionality and quality. In the **operation and maintenance** phases the essential SRE elements are a) continuous monitoring and evaluation of software field reliability, b) estimation of product support staffing needs, and c) software process improvement.

SRE augments all these activities by requiring software developers and maintainers to a) finalize and use operational profiles, b) actively track the development, testing and maintenance process with respect to quality, c) use reliability growth models to monitor and validate software reliability and availability, and d) use reliability-based test stopping criteria to control the testing process and product patching and release schedules.

Ideally, the reaction to SRE information would be quick, and correction, if any, would be applied already within the life-cycle phase in which the information is collected. However, in reality, introduction of an appropriate feedback loop into the software process, and the latency of the reaction, will depend on the accuracy of the feedback models, as well as on the software engineering capabilities of the organization. For instance, it is unlikely that organizations below the third maturity level on the SEI Capability Maturity Model scale [Pau93] would have processes that could react to the feedback information in less than one software release cycle. Reliable latency of less than one phase, is probably not realistic for organizations below level 4 . This needs to taken into account when the level and the economics of SRE implementation is considered .

References

[Abr92] S.R. Abramson et al., "Customer Satisfaction-Based Product Development," Proc. Intl. Switching Symp., Vol. 2. Inst. Electronics, Information, Communications Engineers, Yokohama, Japan, pp. 65-69, 1992.

[AIA93] AIAA/ANSI *Recommended Practice for Software Reliability*, ANSI/AIAA, R-103-1992, American Inst., of Aeronautics and Astronautics, 1993.

[Bel90] BELLCORE, Reliability and Quality Measurements for Telecommunications Systems (RQMS) - TR-TSY-000929, Issue 1, June 1990.

[Boe89] B.W. Boehm, *Tutorial: Software Risk Management*, IEEE CS Press, 1989.

[Bri93] L.C. Briand, W.M. Thomas and C.J. Hetsmanski, "Modeling and Managing Risk Early in Software Development," Proc. 15th ICSE, pp 55-65, 1993.

[Bro92] S. Brocklehurst and B. Littlewood, "New Ways to Get Accurate Reliability Measures," IEEE Software, pp. 34-42, July 1992.

[Chr90] D.A. Christenson, S.T. Huang, and A.J. Lamperez, "Statistical Quality Control Applied to Code Inspections," IEEE J. on Selected Areas in Communications, Vol. 8 (2), pp. 196-200, 1990.

[Coh94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton "The Automatic Efficient Test Generator (AETG) System", Proc. IEEE Int. Symp. Software Reliability Engineering, 1994, pp. 303-309.

[Coh96] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "Combinatorial Design Approach to Test Generation", IEEE Software, Sept. 1996, pp. 83-88.

[Coh97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design, "IEEE Trans. Soft. Eng., Vol. 23, No. 7, July 1997, pp. 437-444..

[Cra92] Cramp R., Vouk M.A., and Jones W., "On Operational Availability of a Large Software-Based Telecommunications System," Proc. Third Intl. Symposium on Software Reliability Engineering, IEEE CS, 1992, pp. 358-366

[Ehr93] W. Ehrlich, B. Prasanna. J. Sampfel, J. Wu, "Determining the Cost of Stop-Test Decisions," IEEE Software, Vol 10(2), pp 33-42., 1993

[Far88] W.H. Farr, "Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) — Library Access Guide", TR84-371 (Rev.1), Naval Surface Warfare Center, Dahlgren VA; also "SMERFS User's Guide," TR84-373 (Rev.1), 1988.

[FCC92] Federal Communications Commission, "Notification by Common Carriers of Service Disruptions," 47 CFR Part 63, Federal Register, Vol. 57 (44), March 5, 1992, pp 7883-7885.

[Fra88] E.G. Frankel, *Systems Reliability and Risk Analysis*, Second Revised Edition, Kluwer Academic Publishers, 1988.

[Goe79] A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and other Performance Measures, " IEEE Trans. on Reliability, Vol R-28(3), pp. 206-211, 1979.

[IEE86] IEEE Std. 1012-1986, IEEE Standard *Software Verification and Validation Plans*, IEEE 1986.

[IEE88a] IEEE Std. 982.1-1988, IEEE Standard *Dictionary of Measures to Produce Reliable Software*, IEEE 1988.

[IEE88b] IEEE Std. 982.2-1988, IEEE Guide *for the use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE 1988.

[IEE90] IEEE Std. 610.12-1990, IEEE Standard *Glossary of Software Engineering Terminology*, IEEE 1990.

[Jon93] Jones, W. D., and Gregory, D., "Infinite Failure Models for a Finite World: A Simulation of the Fault Discovery Process," *Proceedings of the Fourth International Symposium on Software Reliability Engineering*, pp. 284-293, November 1993

[Kan93a] Kanoun K., Kaaniche M., Laprie J-C., and S. Metge "SoRel: A Tool for Software Reliability Analysis and Evaluation from Statistical Failure Data," Proc. 23rd IEEE intl. Symp. on Fault-Tolerant Computing, Toulouse, France, June 1993, pp. 654-659.

[Kan93b] Kanoun K., Kaaniche M., and Laprie J-C., "Experience in Software reliability: From Data Collection to Quantitative Evaluation," Proc. Fourth Intl. Symposium on Software Reliability Engineering, Denver, Colorado, November 3-6, 1993, pp. 234-245.

[Kho90] T.M. Khoshgoftaar and J.C. Munson, "Predicting Software Development Errors Using Software Complexity Metrics," IEEE J. on Selected Areas in Communications, Vol. 8 (2), pp 253-261, 1990.

[Lap91] J.C. Laprie, K. Kanoun, C. Beounes, and M. Kaaniche, "The KAT (Knowledge-Action-Transformation) Approach to the Modeling and Evaluation of Reliability and Availability Growth," *IEEE Transactions on Software Engineering*, IEEE, Vol 18 (4), April 1991, pp. 701-714.

[Lei98] Y. Lei and K. C. Tai, "In-Parameter-Order: A test generation strategy for pairwise testing," Proc. 3rd IEEE

High-Assurance Systems Engineering Symposium, Nov. 1998, 254-261.
[Lyu 94] M.R. Lyu (ed.), Software Fault Tolerance, Trends-in-Software Book Series, Wiley, 1994
[Lyu92] M.R. Lyu and A. Nikora, "Applying Reliability Models More Effectively," IEEE Software, pp. 43-45, July 1992.
[Lyu96] Handbook of Software Reliability Engineering, McGraw Hill, editor M. Lyu, 1996.
[Mal91] Y.K. Malaiya, Editor, *Software Reliability Models: Theoretical Developments, Evaluation and Application*, IEEE CS Press, 1991.
[Mus87] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
[Mus90] J.D. Musa, and W.W. Everett, "Software-Reliability Engineering: Technology for the 1990s," IEEE Software, Vol. 7, pp. 36-43, November 1990
[Mus93] J.D. Musa, "Operational profiles in Software-Reliability Engineering," IEEE Software, Vol. 10 (2), pp. 14-32, March 1993.
[Mus98] J.D. Musa, *Software Reliability Engineering*, McGraw-Hill, New York, 1998.
[Ohb84] M. Ohba, "Software Reliability Analysis Models," IBM J. of Res. and Development, Vol. 28 (4), pp. 428-443, 1984.
[Pau93] M.C. Paulk, B. Curtis, M. B. Chrissis, and C.V. Weber, "Capability Maturity Model, Version 1.1," IEEE Software, pp. 18-27, July 1993.
[Pha92] H. Pham, ed., *Fault-Tolerant Software Systems: Techniques and Applications*, IEEE Computer Society Press, 1992.
[Pot97] Potok, T. and M. Vouk, "The effects of the business model on the object-oriented software development productivity," IBM Systems Journal, Vol. 36(1), pp. 140-161, 1997
[Pre97] R.S. Pressman, 1997 (**Fourth Edition**), *Software Engineering: A Practitioner's Approach*, McGraw-Hill.
[Riv98a] A. Rivers "Software Reliability Modeling During Non-Operational Testing", Ph.D. dissertation North Carolina State University, 1998.
[Riv98b] Anthony T. Rivers and M. A. Vouk "Resource-Constrained Non-Operational Testing Of Software," Proceedings ISSRE 98, 9th International Symposium on Software Reliability Engineering, Paderborn Germany, Nov. 4-7, 1998
[San63] Sandler, G. H., *Systems Reliability Engineering*, Prentice-Hall, Englewood Cliffs, N.J., 1963.
[She97] Sheldon, F.T., *Software Reliability Engineering Case Studies*, 8ht Intl. Symposium on Software Reliability Engineering, IEEE CS Press, November, 1997.
[Sho83] M.L. Shooman, *Software Engineering*, McGraw-Hill, New York, 1983.
[Sin92] N.D. Singpurwalla and R. Soyer, "Nonhomogenous auto-regressive process for tracking (software) reliability growth, and their Bayesian analysis," J. of the Royal Statistical Society, B 54, 145-156, 1992.
[Tri82] K. S. Trivedi, *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
[Vou93] M. A. Vouk and K.C. Tai, "Some Issues in Multi-Phase Software Reliability Modeling," in Proc. CASCON '93, pp. 513-523, October 1993.
[Xie91] M. Xie, *Software Reliability Modeling*, World Scientific, Singapore, New Jersey, London, Hong Kong, 1991.
[Yam83] S. Yamada, M. Ohba, and S. Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," IEEE Tran. on Reliability, Vol. R-32 (5), pp. 475-478, 1983.
[Yam93] S. Yamada, J. Hishitani, and S. Osaki, "Software-Reliability Growth with Weibull Test-Effort: A Model and Application," IEEE Trans. Reliability, Vol. 42(1), pp. 100-106, 1993.

Appendix I - Copies of the Slides

SLIDE-1

Introduction

- Software Reliability Engineering (Musa 1991)

  SRE is the quantitative study of the operational behavior of software-based systems with respect to user requirements. It includes
  (1) software reliability measurement and estimation (prediction),
  (2) the effects of product and development process metrics and factors (activities) on operational software behavior, and
  (3) the application of this knowledge in specifying and guiding software development, acquisition, use, and maintenance.

SLIDE-2

- SRE is "Best Practice" at AT&T.
- It has also been, or is being, adopted by many other leading edge software manufacturers (e.g., Nortel, IBM, NCR, HP, Cray, etc.)
- Now is the optimal time to adopt it.
- SRE is an essential component of Total Quality Management in the context of software.

SLIDE-3

Software Reliability and Availability

- Reliability is one of the most important quality attributes
- Reliability (Fran88)

- The probability that a system or component will perform its intended function for a specified period of "time", under required conditions.

  It can also be defined as the probability that a system, subsystem, or component will give specified performance for the duration of a mission when used in the manner and for the purpose intended, given that the system, subsystem, or component is functioning properly at the start of the mission.

SLIDE-4

- Availability (Fran88)
  - The probability or degree to which a software or an equipment will be ready to start a mission when needed.

    Availability can be expressed as up-time availability, steady state availability, and instant availability.

- Dependability (Fran88)
  - The probability or degree to which an equipment will continue to work until a mission is completed.

SLIDE-5

Value of Software Reliability and Availability Measurement

- Software reliability engineering can guide your decisions and improve performance as a software engineer or manager.

How?

1. It is customer oriented.
2. It is the most important and measurable attribute of software quality.

SLIDE-6

3. It lets you understand the user needs in a quantitative way (tradeoffs, better management).
4. Improves development and operational decisions (specification of design goals, schedules, resources, development management, impact of decisions, control of quality level, etc.)
5. Increases productivity (allows optimization of resources and compliance with customer needs, e.g. test stopping criteria)
6. Improves position of the organization (company) through customer satisfaction, reputation, "market share", "profitability").

SLIDE-7

Cost and Accuracy

- Benefits are far greater than costs.
- Reliability measurement costs are typically about 0.1 - 0.3% of the project cost.
- Benefits (savings) are typically an order of magnitude larger.

SLIDE-8

Software Reliability Measurement and Estimation (Prediction)

- There are many metrics (e.g. collected in IEEE Standards).
- The metrics and models can be classified in many different ways. One possibility is by the life-cycle phase (Rama82): Development (debugging) phase, Validation phase, Operational phase, Maintenance phase, and Correctness measures (test reliability, e.g. error seeding, test coverage measures, etc.)

SLIDE-9

Nelson's Model

Let P{E} denote probability of event E. Then, in most general terms, reliability over i discrete units of exposure period is:

R(i)= reliability over i "runs"

= P{no failure over i "runs"}

Assuming that inputs are selected independently according to some probability distribution function, and faults are static (no corrections take place for the duration of the measurement) we have

$$R(i) = [R(1)^{i}] = R^{i}$$

where R + R(1), i.e. the probability that the program will operate correctly on the next test case and

$$R = 1 - \lim_{n \to \infty} \{n_f/n\}$$

where n is the number of runs, and n_f is the number of failures in n runs.

SLIDE-10

Approximation (estimate of R)

Assuming that inputs are selected independently according to some probability distribution function, and faults are static (no corrections take place for the duration of the measurement) we can estimate the operational software reliability, $\hat{R}$, by

$$\hat{R} = 1 - \frac{n_f}{n}, \quad Var(\hat{R}) = \frac{n_f(n-n_f)}{n^3}$$

where n is the number of testing runs, and n_f is the number of failures in n runs.

SLIDE-11

Confidence Bounds

Approximate confidence bound on this estimate can be obtained by considering the proportion $\frac{n_f}{n}$ and making assumptions about it.

Assuming that the sampling distribution of the estimate $\hat{p} = \frac{n_f}{n}$ is approximately normally distributed and that n>30, then [e.g. see Wal78]: a $(1-\alpha)100\%$ confidence interval for the binomial parameter p is approximately

$$\hat{p} - z_{\alpha/2}\sqrt{\frac{\hat{p}\hat{q}}{n}} < p < \hat{p} + z_{\alpha/2}\sqrt{\frac{\hat{p}\hat{q}}{n}}$$

where $\hat{p}$ is the proportion of successes in a random sample of size n, $\hat{q} = 1 - \hat{p}$, and $z_{\alpha/2}$ is the value of the standard normal curve leaving an area of $\alpha/2$ to the right. For α=0.95, $z_{0.025}$ = 1.96.

SLIDE-12

Continuous Exposure Time

For a continuous exposure variable t, reliability is defined as:

R(t) = reliability over time t = P{no failure in interval [0,t]}

Then

$$R(i) = [R(1)^i] = R^i$$

let the period (0,t] be divided into i segments each of which is t/i in length. Let also the probability that program fails at any instant (of length t/i) be $\lambda t/i$. Then the probability that the program does not fail in the interval (0,t] is approximately

$$R(t) = [1 - \frac{\lambda t}{i}]^i$$

If we let the intervals become smaller and smaller, i.e. i tends to infinity the above expression becomes

$$Lim(i \to infinity) \{[1 - \frac{\lambda t}{i}]^{-i/(\lambda t)}\}^{-\lambda t}.$$

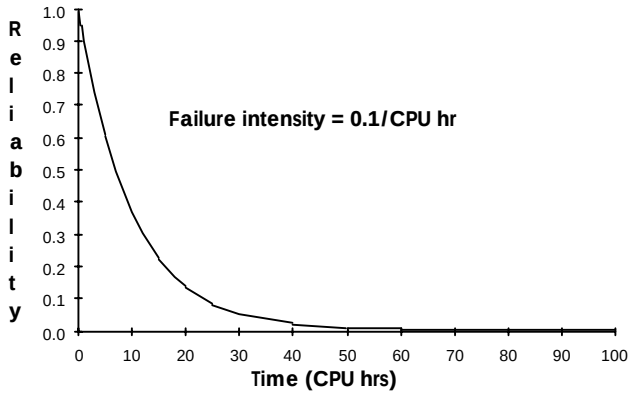
Let $-\lambda t/i = h$, then the above becomes

$$Lim(h \to 0) \{[1 + h]^{1/h}\}^{-\lambda t} = e^{-\lambda t},$$

since the limit in the braces is the common definition of "e". Therefore, in the case of continuous exposure

$$R(t) = e^{-\lambda t}.$$

SLIDE-13



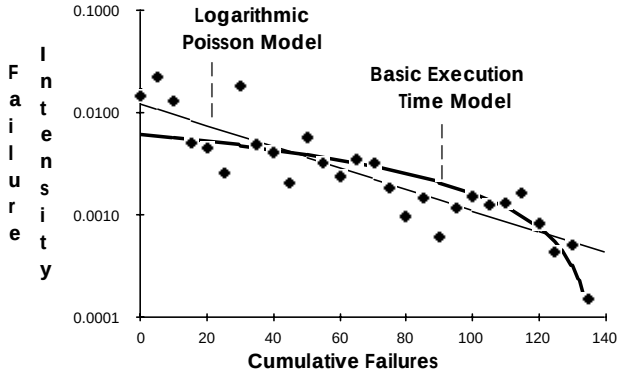
As the testing progresses more test cases are executed and the system response is verified. With successful corrections the estimate of the reliability (R) of the system increases.

SLIDE-14

The Idea of Quality (Reliability) Growth

During testing measure software "failure intensity", or number of failures per unit "time" (time could be CPU minutes or similar) by counting failures and recording times at which failures occur.

Fit an appropriate reliability model.



Use the model to predict future behavior of the effort (how much longer should we test to reach objective failure intensity, how many failures are users likely to experience in the field, etc.).

SLIDE-15

Software Reliability Models

Classification by life-cycle phases [Rama82]:
Development (debugging) phase
Validation phase
Operational phase
Maintenance phase
Correctness measures
(test reliability, error seeding, test coverage measures, etc.)

SLIDE-16

Testing and Debugging Phase

Frequent assumption is that correction of errors does not introduce any new errors. The idea is to increase the reliability of the software (reliability grows). Hence reliability growth models. All models in this class treat the program as a black box. There are two subclasses:

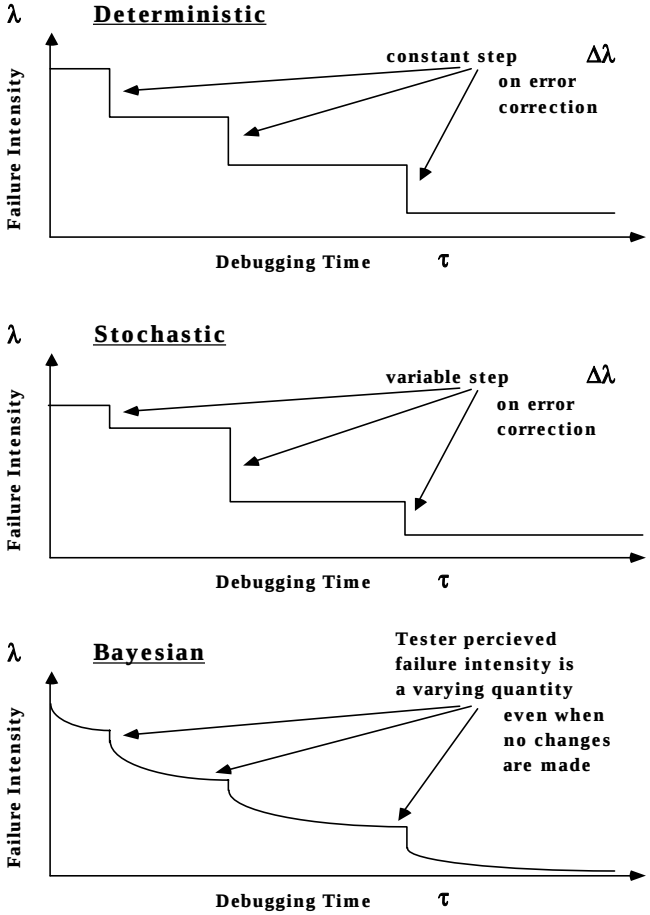
Error counting models:

These models estimate both the number of errors remaining in the program as well as its reliability.

Non-Error Counting Models:

The non-error counting models only estimate the reliability of the software.

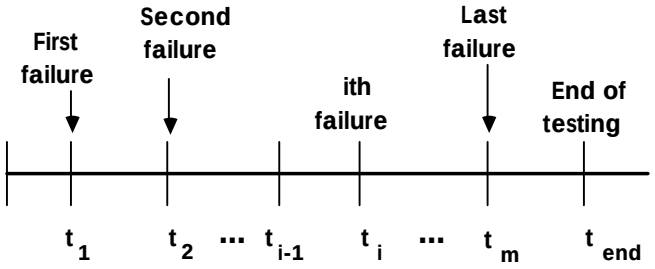
SLIDE-17



SLIDE-18

An Example

Observed failure intensity can be computed in a straightforward manner from the tables of failure time or grouped data (e.g. Musa et al. 1987).



Example: (136 failures total):
Failure Times (CPU seconds): 3, 33, 146, 227, 342, 351, 353, 444, 556, 571, 709, 759, 836 ..., 88682.

Data are grouped into sets of 5 and the observed intensity, cumulative failure distribution and mean failure times are computed, tabulated and plotted.

SLIDE-19

Cum. Failures	Cum. Time	Intensity	Time
5	342	0.014620	171.00
10	571	0.021834	456.50
15	968	0.012594	769.50
20	1984	0.004921	1476.00
25	3098	0.004488	2541.00
30	5049	0.002563	4073.50
35	5324	0.018182	5186.50
40	6380	0.004735	5852.00
45	7644	0.003956	7012.00
50	10089	0.002045	8866.50
55	10982	0.005599	10535.50
60	12559	0.003171	11770.50
65	14708	0.002327	13633.50
70	16185	0.003385	15446.50
75	17758	0.003179	16971.50
80	20567	0.001780	19162.50
85	25910	0.000936	23238.50
90	29361	0.001449	27635.50
95	37642	0.000604	33501.50
100	42015	0.001143	39828.50
105	45406	0.001474	43710.50
110	49416	0.001247	47411.00
115	53321	0.001280	51368.50
120	56485	0.001580	54903.00
125	62661	0.000810	59573.00
130	74364	0.000427	68512.50
135	84566	0.000490	79465.00
136	88682	0.000243	86624.00

$$\text{Intensity (failures/CPU sec)} = \frac{\Delta f}{\Delta t} = \frac{5}{\text{Cum.T}_2 - \text{Cum.T}_1}$$

$$\text{Time (average)} = \text{Cum.T}_1 + \frac{\Delta t}{2}$$

Two common models are the "basic execution time model" and the "logarithmic Poisson execution time model" (e.g. Musa et al. 1987).

SLIDE-20

Basic Execution Time Model

Failure intensity $\lambda(\tau)$ with debugging time τ:

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

where λ_0 is the initial intensity and ν_0 is the total expected number of failures (faults). Also

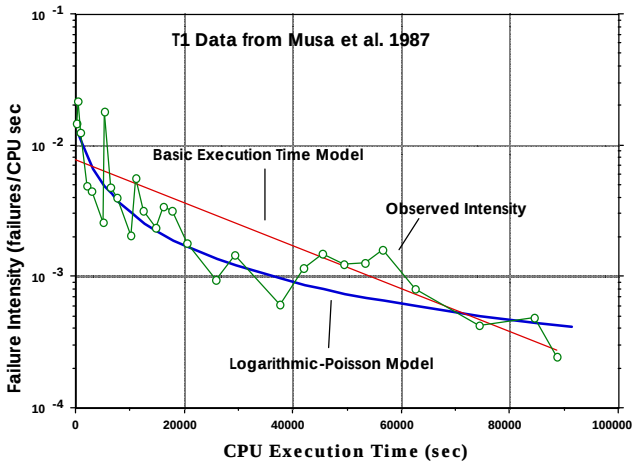
$$\lambda(\tau) = \lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right)$$

where $\mu(\tau)$ is the mean number of failures experienced by time τ.

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

SLIDE-21

An Example: T1 Data from Musa et al 1987.



- In this case the Logarithmic-Poisson Model fits somewhat better than the Basic Execution Time Model. In some other projects BE model fits better than LP model.

SLIDE-22

Some Derived Information

Additional expected number of failures, $\Delta\mu$, that must be experienced to reach a failure intensity objective

$$\Delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F),$$

where λ_P is the present failure intensity, and λ_F is the failure intensity objective. The additional execution time, $\Delta\tau$, required to reach the failure intensity objective is

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln\frac{\lambda_P}{\lambda_F}$$

SLIDE-23

- After fitting a model describing the failure process we can estimate its parameters, and the quantities such as the total number of faults in the code, future failure intensity and additional time required to achieve a failure intensity objective.

- It is extremely important to compute confidence bounds for any estimated parameters and derived quantities in order to see how much can one rely on the obtained figures.
- Accurate and comprehensive data collection is of ultimate importance.
- For example, The failure data should include
  - the times of successive failures (alternatively intervals between failures may be collected),
  - or the number of failures experienced during an interval of testing (grouped data),
  - information about each corrected fault,
  - information about the parts of the code affected by the changes,

etc.

SLIDE-24

Availability

- The probability or degree to which a software or an equipment will be ready to start a mission when needed.

Steady state availability. The system becomes independent of its starting state after operating for enough time. This steady-state availability of the system is

$$A_{SS} = \frac{\mu}{\mu+\lambda}$$

Where μ is the system repair rate (failures repaired per unit time) and λ is the system failure rate (failures per unit time). Both are assumed constant.

SLIDE-25

Mean Time to Failure and Repair

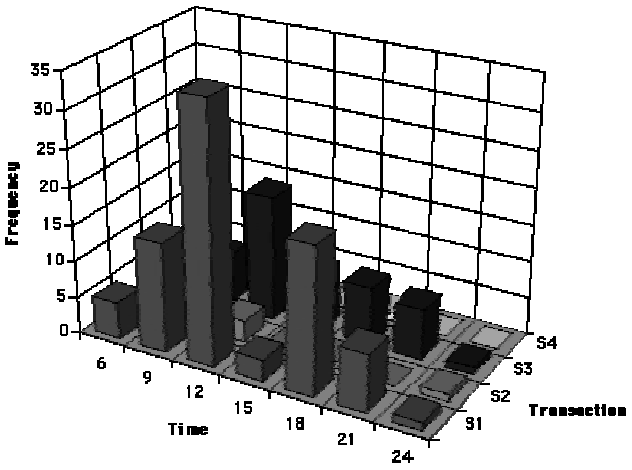
- Mean time to failure = MTTF = $\frac{1}{\lambda}$
- Mean time to repair = MTTR = $\frac{1}{\mu}$

$$A_{SS} = \frac{MTTF}{MTTR+MTTF}$$

SLIDE-26

Operational Profile

- The crucial concept is the operational profile.
- Operational profile (OP) is the set of relative frequencies of occurrence of the run types, usually expressed as fractions of the total of runs (in which case, we have probabilities).

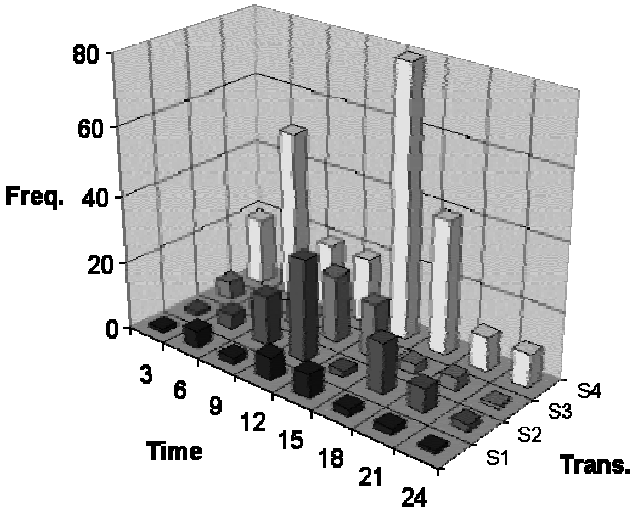


- The OP is used to guide testing, but it can be also employed to guide managerial and engineering decisions throughout the software life-cycle by highlighting most important alternatives (prioritized operational profile).
- The absolutely essential step in applying "Operational Testing" is the definition of the profile itself.

SLIDE-27

Risk-Based Profile

- Risk-based profiles combine the usage pattern with the cost or loss factors into a risk-profile.
- Risk = Probability_of_unsatisfactory_event * Cost_or_Loss_magnitude.



SLIDE-28

Procedure

- Participants: system engineers, high-level designers, test planners, product planning, marketing.
- Process: Iterative, converging.

- Determine the profile by
  - Define the customer profile
  - Establish the user profile
  - Define the system-mode profile (number of profiles needed)
  - Determine the functional profile(s)
  - Determine the operational profile(s)
- The process starts during the requirements phase and continues until the system testing starts.

SLIDE-29

- Profile: A set of disjoint alternatives with probability of occurrence attached.

Alternative	Probability
A	0.3
B	0.6
C	0.1

  - If usage is available as a rate (e.g., transactions per hour), it needs to be converted to probability by dividing by the total number of transactions per hour.
  - Probability may not be the only criterion for choosing the profile. Cost of failure (severity, importance) of that operation plays a role. Generate profiles for each category of criticality (typically four separated by at least an order of magnitude in effects).

SLIDE-30

Customer Profile

- Customer: person, group, or institution that ACQUIRES the system.

Telephone Switch

Customer group	Probability
Local Carrier	0.7
Inter-City Carrier	0.3

SLIDE-31

User Profile

- Customer: person, group, or institution that EMPLOYS the system.

User group	Local (0.7)		Inter-City (0.3)		Total
	within	total	within	total	
Households	0.6	0.42	0.2	0.06	0.48
Businesses	0.3	0.21	0.6	0.18	0.39
Emergency Services	0.05	0.035	0.001	.0003	.0353
Other	0.05	0.035	0.199	.0597	.0947

SLIDE-32

Mode Profile

- System mode is a set of functions or operations that are grouped for conveniences in analyzing execution behavior. System can switch among modes, or two or more can be active.

- Determine operational profile for each system mode.

- Some bases for mode classification:
  - User groups
  - Environment (e.g., overload vs. normal, high-noise vs. low-noise)
  - Criticality
  - User experience
  - Platform
  - How user employs system operations to accomplish system functions.

SLIDE-33

Assume (voice: inter-city household 99%, local household 90%, business 70% both, System Administration 30% of Other, and Maintenance 70% of Other, rest data)

Mode		Probability
Voice (personal)	0.4374	*0.42*0.9+0.06 *0.99*
Voice (business)	0.273	*0.39*0.7*
Data	0.1596	*0.42*0.1 + 0.06*0.01 + 0.3*0.39*
Emergency	0.0353	0.0353
System Admin.	0.02841	*0.0947*0.3*
Maintenance	0.06629	*0.0947*0.7*

SLIDE-34

Functional Profile

- Break each system mode into user-oriented functions (features + environment) needed for its operation.
- Generate function list, find probability of function occurrence (relative use of functions within that profile). A function may be composed of one or more tasks,

operations, and parameters and environmental variables (inputs).
- Keep the number of functions in the range 50 to several hundred.

X(A,B)
with
A = A1 or A2
B = B1 and B2

- Add environmental variables (e.g., platform, analog/digital device, operating system). Probability of occurrence.

SLIDE-35

- Possible functions:

X or
{X(A1,B), X(A2,B)} or
{X(A1,B1), X(A2,B1), X(A1,B2), X(A2,B2)}

- Implicit format example:

X(A,C), {A1,A2}, {C1, C2, C3}

Parameter Value	Probability
A1	0.1
A2	0.9
C1	0.5
C2	0.1
C3	0.4

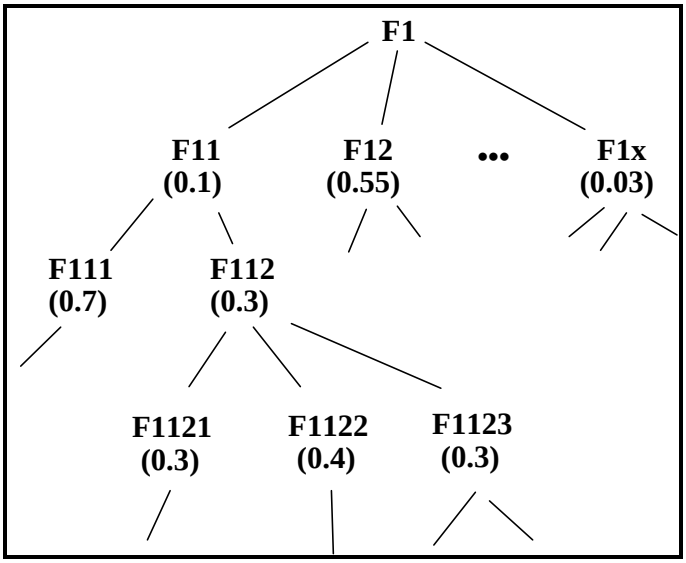
- Explicit format example:

{X(A1,B1), X(A2,B1), X(A1,B2), X(A2,B2)}

Parameter Value	Probability
A1*C1	0.05
A1*C2	0.01
A1*C3	0.04
A2*C1	0.45
A2*C2	0.09
A2*C3	0.36

SLIDE-36

Call Tree Approach



Functions for Mode α	Environment	Probability
F1	A	
F11	A	
F11	B	
F12	A	
F12	B	

SLIDE-37

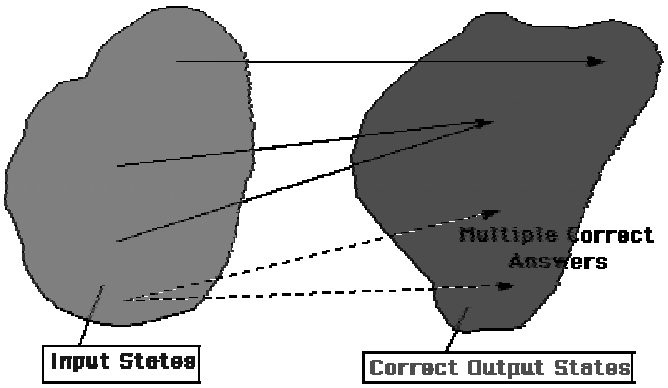
Operational Profile

- Profile of operations that implement functions (features) that users use.
- Functions EVOLVE into operations during system development.
- Operations are the ones tested. Their profile determines testing resources, test cases and the order of their execution.
- Associate operations with run types (execution segment associated with a user-oriented task, e.g., command in a particular environment).

SLIDE-38

Input States

- Identify run input states, and system input space.
- Partition input space, get partition probabilities. States will be selected from these partitions according to these probabilities.



SLIDE-39

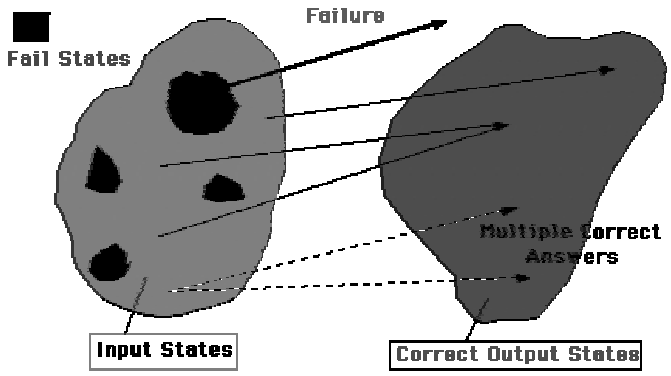
Test Selection and Testing

- Select operation according to their occurrence probability or usage level in the call tree.
- Partition operations into run categories (if possible) - allowed/meaningful combinations of parameters (e.g., two at a time). Select randomly or according to experimental design..
- Randomly select a particular run.
- Pay special attention to special issues (e.g., safety).
- Control (limit) the number of operations.
- Update profile on a regular basis (monitor the field usage).

SLIDE-40

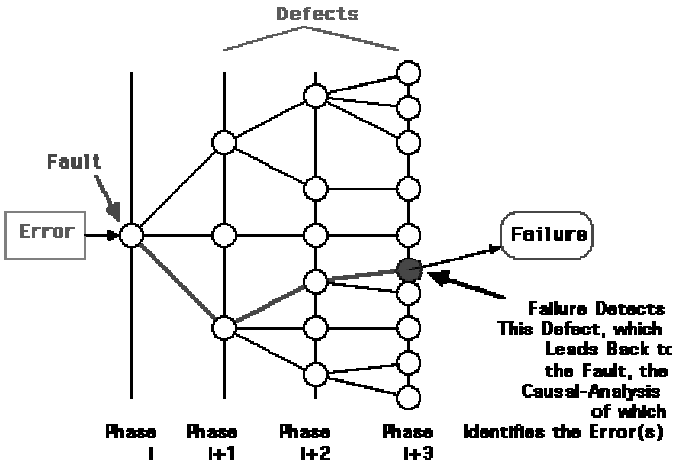
Fail-States

- Pay attention to clustering of **fail-states** and perform root-cause analysis.



SLIDE-41

Root-Cause Analysis



SLIDE-42

Regression Testing

- Do not focus only on the changed code.
- Augment testing of the changed operations with a general operational profile test. It is an efficient way of accounting for the possible influence of the changes on other than the modified operations.
- Pay special attention to special issues (e.g., safety).
- Keep a record of detected problems and analyze it for trends, special areas of concern, etc.

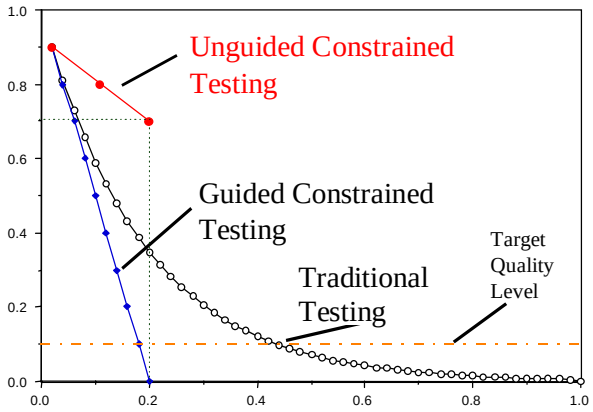
SLIDE-43

Design for Testability

- Design the system so that the control (limit) of the number of operations is possible (e.g., if system modes do not interact then they can be tested separately and minimum interaction testing needs to be done). Good design practices include minimization of the coupling and maximization of the coherence of software modules (object-oriented approach).
- Design system to limit number of environmental variables (cases).
- Pay special attention to special issues (e.g., safety) - they have to be accounted for and TESTABLE.
- Design the system to accommodate testing by approaches such as the statistical experimental design.

SLIDE-44

Constrained Testing



SLIDE-45

Pairwise Testing (1)

Parameters A, B, C, D and E which have the following valid values

A	B	C	D	E
A1	B1	C1	D1	E1
A2	B2	C2	D2	E2
A3	B3		D3	

will require 3x3x2x3x2 = 108 test cases if all three parameters are related (interacting) and we wish to cover all combinations of the parameters.

SLIDE-46

Pairwise Testing (2)

Pairwise testing strategy requires only 11 (eleven) tests to cover all PAIRS of combinations at least once.

Case	A	B	C	D	E
1	A1	B1	C1	D1	E1
2	A1	B1	C1	D1	E2
3	A3	B1	C1	D3	E2
4	A3	B3	C1	D1	E2
5	A2	B2	C1	D1	E2
6	A3	B2	C2	D1	E1
7	A1	B2	C2	D3	E1
8	A2	B3	C2	D3	E1
9	A1	B3	C1	D2	E1
10	A3	B2	C1	D2	E1
11	A2	B1	C2	D2	E2

Examples of some other choices

- Suppose D and E are new paramters and one wants to run a **regression test** only. Generate test according to relation {D,E}. - only **six (6)** tests. This choice focuses on checking interactions between new parameters only.
- Test set according to relation {D,E} and other relations involving some new and old parameters. This is limited checking of interactions between new and ol parameters.
- Delete some values of parameters A, B and C and then generate a test set according to relation {A,B,C,D,E}. This is another way of checking limited interactions between new and old parameters.

SLIDE-47

PairTest
(http://renoir.csc.ncsu.edu/Tools/PairTest)

PairTest is a software tool that generates a test set satisfying the pairwise testing strategy for a system [Lei98}. The major features of PairTest include the following:

- *"PairTest supports the generation of pairwise test sets for systems with or without existing test sets and for systems modified due to changes of input parameters and/or values.*
- *PairTest provides information for planning the effort of testing and the order of applying test cases.*
- *PairTest provide a graphical user interface (GUI) to make the tool easy to use.*
- *PairTest is written in Java and thus can run on different platforms.*
- *The PairTest tool was developed by Dr. K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University."*

SLIDE-48

Relationship with TQM

- TQM principles include
  customer satisfaction,
  quality culture,
  improvement of processes, education and training (job skills and TQM tools),
  defect prevention instead of reactive elimination,
  use of data and statistical tools,
  team approach (both intra- and inter-departmental, and hierarchical), and
  commitment to continuous improvement.

SLIDE-49

- SRE is an integral part of TQM because SRE activities, tasks and techniques are part of TQM.
- Software Maturity Index (SEI)
  The higher the software process maturity of an organization the more elements of TQM (and SRE) need to be incorporated into its process model. Software

maturity index is a measure of extent to which TQM has been permeated the software process. Certain maturity must be present before SRE can be fully implemented.

SLIDE-50

Software Maturity Levels

Level	Description	Problem Areas
1: Initial	Poorly defined procedures and controls; ad hoc process. The organization operates without formalized procedures, cost estimates, and project plans. Even when plans and controls exist there is no management mechanisms to ensure they are followed. Tools are not well integrated with the process, and are not uniformly applied. Change control is lax and senior management is not exposed to or does not understand the key software problems and issues.	Project management, project planning, configuration management, software quality assurance, use of modern tools and technology.
2: Repeatable	Basic project controls have been established. Organization has experience with similar projects, but faces sizable risks when doing novel projects. Quality problems are frequent and framework for orderly improvement is lacking. Software fault data are being collected in a standardized and formal fashion.	Training, technical practices (reviews, testing), process focus (standards, process groups).
3: Defined	Commitment to software process evaluation and improvement through establishment of a software process group(s) and appropriate process architecture(s). There is mostly qualitative understanding of the needed process. Appropriate software engineering standards, methods and technologies are in place.	Process measurement, process analysis, quantitative quality plans.
4: Managed	Process is quantified. Measurements for the quality and productivity of each key task. Process database, analysis and dissemination of process related information (e.g. process efficiency). Errors can be predicted with acceptable accuracy.	Changing to new technology, problem analysis, problem prevention.
5: Optimizing	Process improvement feed-back and feed-forward controls are established. Rigorous defect causal analysis and defect prevention. Proactive management.	Automation

SLIDE-51

Summary

Software Reliability Engineering is the quantitative study of the operational behavior of software-based systems with respect to user requirements. It includes

(1) software reliability measurement and estimation (prediction),
(2) the effects of product and development process metrics and factors (activities) on operational software behavior, and
(3) the application of this knowledge in specifying and guiding software development, testing, acquisition, use, and maintenance (this includes the association with the **business model**).