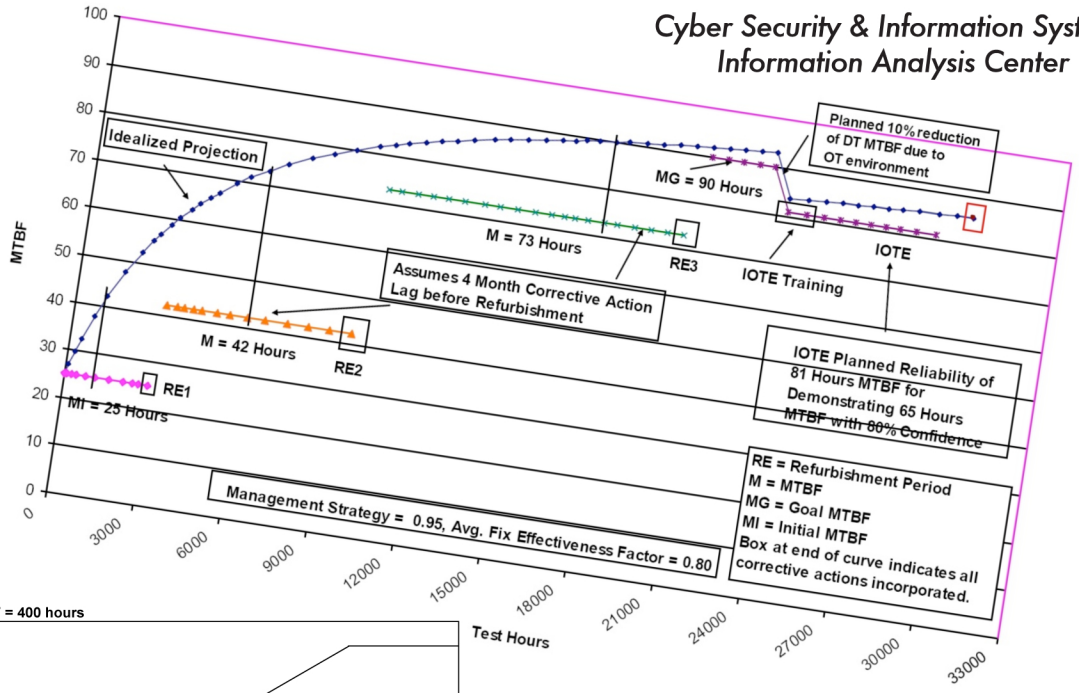
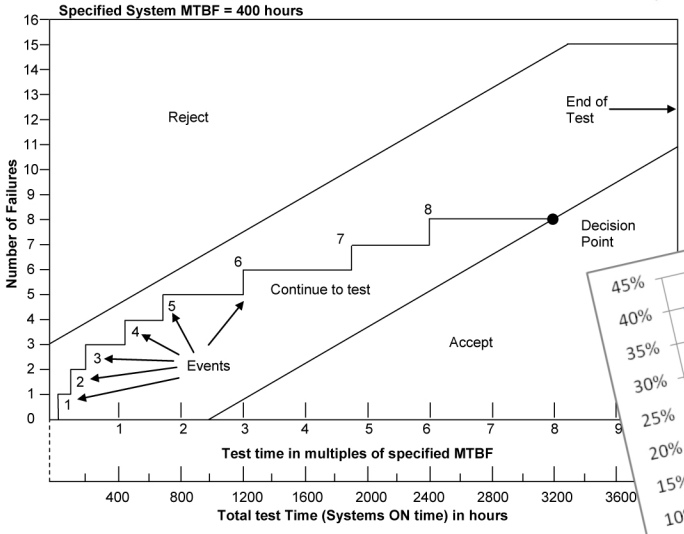
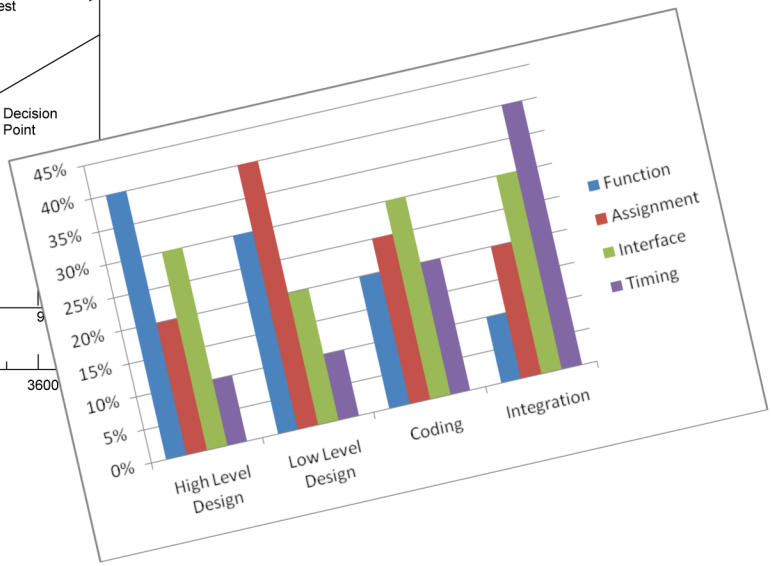
Cyber Security & Information Systems
Information Analysis Center







Handbook of Software Reliability and Security Testing

A CSIAC State of the Art Report

CSIAC is a DoD Information Analysis Center sponsored by the Defense Technical Information Center

Handbook of Software Reliability and Security Testing

A CSIAC State of the Art Report

CSIAC Report Number 519193

Contract FA8075-12-D-0001

Prepared for the Defense Technical Information Center

Prepared By

Taz Daughtrey
Richard Wisniewski
David Nicholls
Thomas McGibbon

Cyber Security and Information Systems Information Analysis Center
Quanterion Solutions Inc.
100 Seymour Road, Suite C102
Utica, NY 13502-1311

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(23-05-2011)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
15-01-2013	Technical	N/A

4. TITLE AND SUBTITLE
Handbook of Software Reliability and Security Testing

5a. CONTRACT NUMBER
FA8075-12-D-0001

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Hugh "Taz" Daughtrey, Richard Wisniewski, David Nicholls, Thomas McGibbon

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cyber Security and Information Systems Information Analysis Center
Quanterion Solutions, Inc., 100 Seymour Rd Suite C102
Utica, NY 13502-1311

8. PERFORMING ORGANIZATION REPORT NUMBER
DAN #519193

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Technical Information Center
DTIC/AI
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060

10. SPONSOR/MONITOR'S ACRONYM(S)
DTIC

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release, Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The level of reliability achieved by a DoD system is instrumental to its level of success in today's demanding missions with increasingly limited resources. Each organization must make its own determination as to why and how much reliability is necessary in its designs, as a direct function of what its customer explicitly or implicitly demands. Factors to be considered in the commercial world include the characteristics of the marketplace, the cost of implementing or not implementing a reliability program strategy; and complete knowledge and understanding of customer expectations. As systems, and systems-of-systems, become increasingly complex, the realm of reliability necessarily extends beyond consideration of solely hardware reliability. It must expand to address significant issues pertaining to software reliability and security (the subject of this Handbook) and human factors as a function of human-machine interactions.

15. SUBJECT TERMS
Software Reliability, Security, Cyber Security, Software Engineering, Software Testing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Thomas McGibbon
U	U	U	UU	340	**19b. TELEPHONE NUMBER** *(include area code)* 315-351-4203

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

Section 1.0: Need for Software Reliability

INTRODUCTION

The level of reliability achieved by a DoD system is instrumental to its level of success in today's demanding missions with increasingly limited resources. Each organization must make its own determination as to why and how much reliability is necessary in its designs, as a direct function of what its customer explicitly or implicitly demands. Factors to be considered in the commercial world include the characteristics of the marketplace (market growth, competitors' strategies, etc.); the cost (in dollars and opportunities) of implementing or not implementing a reliability program strategy; and complete knowledge and understanding of customer expectations, and whether they can be realistically met. Achieved reliability may be relatively simple to quantify (i.e., warranty experience, MTBF), but it will also impact qualitative issues which the organization must be willing to address in order to succeed:

- Manufacturer liability
- Customer perceptions and expectations, specifically in response to security breaches
- Market competition
- Diverse market needs
- Strategies for competitive advantage
- Regulatory requirements for security precautions

For military systems, a unique set of needs exist that must be adequately addressed in order to satisfy operational readiness requirements in an inherently hostile environment, including extended useful life; emphasis on safety, support and operational factors; purchase decision criteria vs. meeting military market needs; and the need for systems and products to operate (and interoperate) under a variety of adverse environmental and operational conditions including adversaries who attempt to degrade or misuse systems. As systems, and systems-of-systems, become increasingly complex, the realm of reliability necessarily extends beyond consideration of solely hardware reliability. It must expand to address significant issues pertaining to software reliability and security (the subject of this Handbook) and human factors as a function of human-machine interactions.

The purpose of Section 1.0 is to highlight those high-level considerations which are linked to both military system and commercial product reliability, recognizing that the use of NDI, COTS, GOTS and other commercial open sources can and will strongly impact the reliability of military systems.

Topic 1.1: How to Use This Handbook

Much of the content in this document may be helpfully accessed by going directly to a specific topic of interest. However, some fundamental discussions appear in the opening topics, so it is recommended that the reader examine all of Sections 1 and 2 as a proper framework before referencing detailed material in the rest of the Handbook. "For More Information" listings at the end of each topic provide additional sources to be consulted, whether for basic background or more advanced treatment of concepts. These references are numbered separately within each topic, so any topic can be read in isolation.

Section 3 details a range of testing types, with Topic 3.1 indicating in tabular form specific characteristics of each approach. Some of the additional suggestions below are drawn from syllabi posted by the International Software Testing Qualifications Board (www.istqb.org).

Possible test selection criteria:

>> Do you have visibility into the software design and implementation?
IF you do have visibility ("glass box"), these testing topics can apply:
Control Flow (3.5.3)
Loop (3.5.4)
Data Flow (3.5.5)
Transaction Flow (3.5.6)
ELSE you do not have visibility ("black box"), and these testing topics can apply:
Finite State (3.5.8)
Orthogonal Array (3.5.9)

>> Do you have usage data or models for distribution patterns of system use?
IF you do have usage data or models, these testing topics can apply:
Statistical Usage (3.5.10)
Operational Profile (3.5.11)
Markov (3.5.12)

>> Were systems specifications provided in formal (algebraic) notation?
IF you have formal specifications, this testing topic can apply:
Domain (3.5.7)

>> Are you concerned about dynamic behavior such as unreachable code, badly designed loops, misdirected function calls, or incorrect sequencing of operations?
IF you are seeking dynamic anomalies, then consider this testing topic:
Control Flow (3.5.3)

>> Are you concerned about data anomalies. such as performing incorrect actions on data variables or performing the correct action on a variable at the wrong time?
IF you are seeking data anomalies, then consider this testing topic:
Data Flow (3.5.5)

>> Are you concerned about incorrect or unsupported transitions between system states, states with no exits, or missing states?
IF you are seeking processing anomalies, then consider this testing topic:
Finite State (3.5.8)

Security testing is especially driven by developments not only in technology but also by evolving threats and exploit mechanisms. A number of relevant reference documents are included in the CD version of this product, and an ongoing effort will provide timely updates to registered users.

Topic 1.2: The Need for Software Reliability

Reliable software does what it is supposed to do. Unreliable software fails to meet expectations, but may do so in any of a number of ways. An unreliable software-based system may be unavailable, incorrect, vulnerable, or possibly even unsafe. This variety of inadequacies and failure modes includes both "sins of omission" (not behaving as intended) and also "sins of commission" (behaving in unintended ways).

Failures of safety-critical systems imperil safety. Failures of mission-critical systems frustrate the accomplishment of their entrusted missions. Systems handling sensitive personal or financial information can have security-breaching failure modes.

Failure of a software-based system, as with any other system, arises from defects (also called faults) in the system. In turn, these defects are caused by human errors or mistakes. The unacceptable behavior known as a failure is the visible manifestation of a previously undetected deviation from intention.

Software unreliability arises from errors such as incomplete, ambiguous or conflicting requirements specification and inappropriate design choices. These errors produce defects that may often be subtle and very difficult to locate, given software's complexity and immateriality.

The subset of defects that may be exploited to breach security are typically referred to as *vulnerabilitie*s. The overall reduction of errors and thus a lower rate of defect injection tend to correspondingly reduce such vulnerabilities. Therefore the following discussions of reliability apply to security issues as well as other manifestations of unreliability. Where appropriate, security-specific topics or techniques will supplement the general treatment of software reliability.

A major prerequisite for determining overall system reliability requirements and specific design reliability requirements at lower levels of system indenture is dependent on a good understanding of the overall operational environment; i.e., the geographical location where the system is likely to be deployed and utilized, the nature and culture of the operating agency (organization), the availability of appropriate technologies and tools, the system procurement and acquisition process, the corporate or political structure, potential threats to data confidentiality, integrity, or availability, and so on. Additionally, it should be recognized that system development is highly dynamic, and the need for agility and flexibility in system design is a critical factor in ensuring that reliability requirements are met.

Although individual perceptions as to today's challenges will differ depending on personal experiences and observations, there are a number of trends that appear to be significant. Some of these trends are noted as follows:

1. *Constantly changing requirements.* The requirements for new systems are constantly changing due to worldwide dynamic conditions, changes in actual and perceived mission threats and priorities, and the introduction of new technologies on an evolving basis. Thus, there is a need for an *open-architecture* approach in the design of systems, and for a highly flexible (agile) systems support capability.
2. *More emphasis on systems.* There is greater emphasis on total *systems* and *systems-of*-systems versus the *components* of systems. The system needs to be addressed in total, and throughout its entire life cycle, to ensure that the necessary functions are being accomplished in an effective and efficient manner. Thus, the reliability design and support infrastructure must be considered a major element of the system, it must be in place and reliable, and it must be readily available to impact the prime mission-related elements of the system. The requirements pertaining to *design for reliability* must address the prime mission-related elements of the system.
3. *Increasing system complexities.* The structures of many systems are becoming more complex as new technologies are introduced and evolve. The system must be designed such that changes can be incorporated quickly, efficiently, and without significantly affecting its overall configuration. Given these constraints, the reliability design and support infrastructure must address the added complexities.

4. *Deliberate attempts to misuse systems.* No longer do systems simply fail "on their own" because of inherent flaws. Increasingly – and for a variety of motivations – unauthorized users are seeking to misuse computing resources to the detriment of their intended missions. Security concerns are typically expressed along several dimensions, such as confidentiality, integrity, and availability. Each aspect must be considered in terms of the priorities of the stakeholders. For instance, a denial-of-service attack might diminish availability without compromising data integrity or confidentiality.

5. *Extended system life cycles -- shorter technology life cycles.* The life cycles of many current systems are being extended for any number of reasons, while, at the same time, the life cycles of most technologies are becoming relatively shorter (due to obsolescence). It is necessary to design systems such that a new technology can be incorporated easily and efficiently. At the same time, the reliability design and support infrastructure must remain responsive, and the duration (life cycle) of the reliability program is likely to be longer due to the extended system life cycles.

6. *Greater utilization of commercial off-the-shelf (COTS) products and Open Source Software (OSS).* With defense system goals continually focusing on lower initial costs and shorter and more efficient procurement and acquisition cycles, there has been more emphasis on the utilization of best commercial practices, standard processes, and commercial off-the-shelf (COTS) equipment and software. Note that COTS issues and concerns equally apply to any Non-developmental Item (NDI) such as Government-off-the-Shelf (GOTS), etc. As a result, there is a greater need for a good definition of requirements at the outset, and there is a greater emphasis on the design of systems (and their major subsystems) versus the design of components. Much of the required reliability program activity has shifted from a major producer to one or more suppliers. This shift, in turn, has increased the complexity of the overall reliability program network, with more organizations participating, as well as some added challenges in determining detailed reliability requirements for many of the COTS and OSS items being utilized in various military system design configurations.

7. *Increasing globalization.* The world is shrinking and there is more trading with (and dependency on) manufacturers and suppliers throughout the world. This has been facilitated through rapid and improved communication practices, the availability of quicker and more efficient packaging and transportation methods, the application of electronic commerce (EC) methods for expediting the accomplishment of procurement and related processes, and so on.

8. *More outsourcing.* There is more outsourcing and the procurement of COTS/OSS items (equipment, software, processes, services) from external sources of supply than ever before. Thus, there are more suppliers associated with almost any given program. Consequently, there must be greater emphasis on the early definition and allocation of system-level requirements, the development of a good and complete set of specifications, and a closely coordinated and integrated level of activity throughout the system development and acquisition process. At the same time, a well-integrated reliability program capability must be developed and implemented when required. This can best be accomplished through the effective implementation of the system engineering process and the proper specification of *design for reliability (DFR)* requirements from the beginning.

9. *Greater international competition.* Along with the noted trends toward increasing globalization and more outsourcing, there is more international competition than ever before, owing not only to improvements in communications and transportation methods, but to the greater utilization of COTS/OSS items and the establishment of effective partnerships worldwide. A major goal is, of course, to deliver in a short time frame a product and/or service that is highly reliable, high quality, cost-effective, and with complete customer satisfaction in mind.

10. *Higher overall life-cycle costs.* In general, experience has indicated that the life-cycle costs of many current systems are increasing. Whereas much emphasis has always been placed on minimizing the costs associated with the procurement and acquisition of systems, relatively little attention has been dedicated to the costs of system operation and support until recent years. In designing systems, one needs to view all decisions in the context of the *total* cost to properly assess the risks associated with the decision(s) in question. As the reliability design and support infrastructure is a major element of the system, and often represents a high-cost contributor, the various alternative approaches in the design of such must be justified

on the basis of total life-cycle cost. Thus, design for reliability must consider not only the "technical" characteristics of design but the "economical" aspects as well.

It should be emphasized that these trends are all interrelated and need to be addressed as an integrated set when determining the requirements for systems and for properly tailoring the reliability programs necessary to support those systems. Further, an awareness of these issues associated with the environment is essential in the *design for reliability*.

Although some of the foregoing and related trends have evolved over time, the tendency is to ignore the changes that have taken place and continue with a business-as-usual approach by implementing past practices, many of which tend to inhibit innovation and growth. Since the operating environment has undergone a major transition in recent years, the requirements for reliability design and support have also undergone significant changes (e.g., the increased emphasis on software and human reliability, as opposed to strictly hardware reliability), and it is anticipated that such changes will continue to evolve.

Section 2.0: Software and System Reliability

INTRODUCTION

In order for reliable systems to be produced, there needs to be a fundamental understanding of the systems engineering (SE) process and an appreciation that the reliability, maintainability and quality disciplines must be an integrated part of that process, beginning with the earliest stages of system concept development. The focus of the reliability engineering process within the systems engineering process can no longer be only on the piece parts and hardware components. As manufacturing and materials technology has significantly improved, so has the inherent reliability of hardware. Systems continue to become much more complex, however, with a greater percentage of their functionality being accomplished by software. As a result, software reliability plays a much more critical role in the ability of a system to successfully meet its mission objectives.

Topic 2.1: Overview of Software Reliability Engineering

The software reliability engineering discipline is relatively young, having germinated in the mid-1970's when it was thought that the software development environment was reasonably stable. An initial wave of software models was the first attempt to bring quantitative reliability measures to the software engineering discipline. This stability, ironically, was short-lived, and a surge of new technology, new paradigms, new structured analysis concepts, and new software development models emerged and continue to shape the growth and evolution of the software reliability process. Figure 2.1-1 chronologically highlights some of the major elements that have simultaneously improved and complicated the development of products that strive for acceptable levels of software reliability.

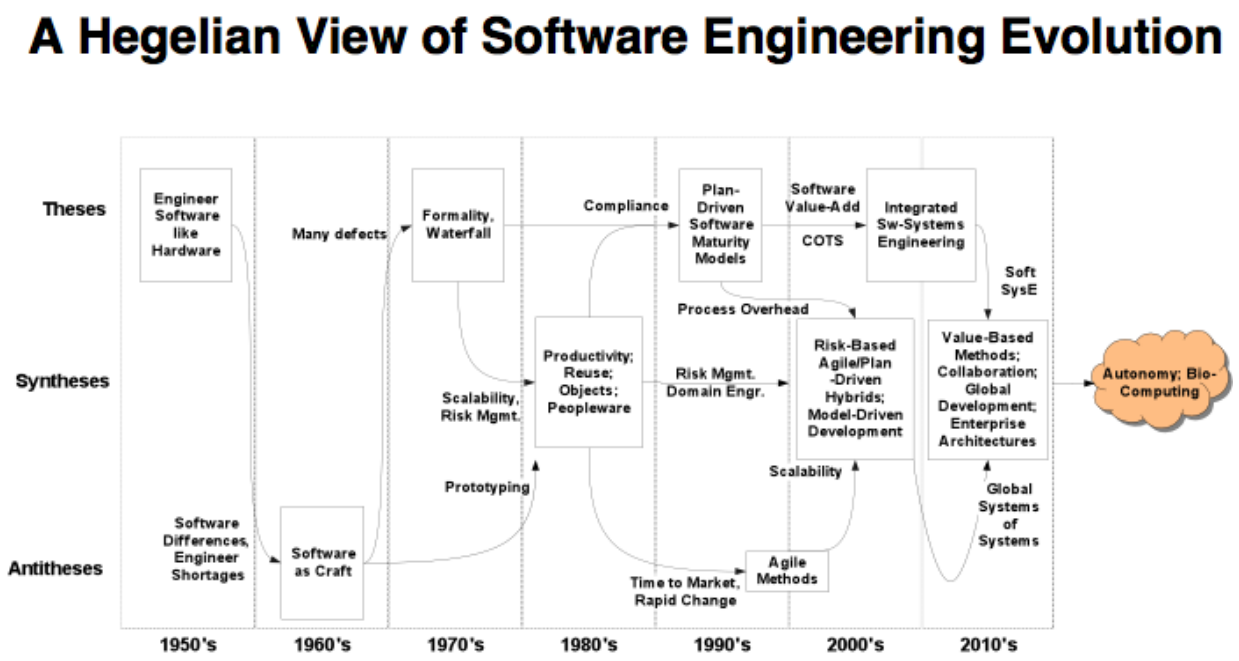


Figure 2.1-1: A View of 20th and 21st Century Software Engineering [Barry Boehm, Keynote Address, 2006 International Conference on Software Engineering]

Why is there growing emphasis on achieving increasingly higher levels of software reliability? Perceptions about the need for highly reliable software have changed, as more and more products and systems depend upon software in order to help meet the needs of the marketplace for "smaller, better, and faster". As a result, software is exercising increasing control over our personal lives (never mind our professional lives) on a daily basis:

- Housework (dishwashers, ovens, etc.)
- Transportation (personal automobiles and mass transit)
- "Creature comforts" (HVAC and lighting controlled by computers)
- Finances (direct deposit, E-commerce, automated billing, etc.)
- Entertainment (video games, audio/video components, etc.)

With this control, however, comes potential customer dissatisfaction (if the controls don't work reliably) or product liability (if the controls don't work reliably and someone gets hurt/killed when they fail), either of which can result in loss of revenue, decreased market share and unfavorable customer perceptions. According to Musa (Reference 7), surveys of users of software-based systems indicate that on the average reliability/availability, rapid delivery and low cost (in that order) are regarded as the most important quality characteristics. Figure 2.1-2, from

http://gizmodo.com/5868029/the-worst-computer-bugs-of-2011, enumerates significant losses from just the most recent year.

"The Most Expensive Computer Bugs of 2011"

Earlier this year a man lost a $57 million jackpot when a casino alleged a "software glitch" on the slot machine. Well, that's nothing compared to the backlog of $9 billion in unprocessed payments that happened in Japan in March.

Here is the top five *worst*, most expensive computer glitches of 2011, according to SQS, a UK company specialized in software quality assurance:

1. Financial firm services AXA Rosenberg lost $217 million of its investors' money because of a software glitch in its investment model. The company hid the bug from its clients, so they had to pay back that amount — plus a $25 million fine — to the US Securities and Exchange Commission

2. Car manufacturer Honda had to recall 2.5 million cars because of a bug that allowed vehicles to shift out of park or simply stall out

3. Japanese bank Mizuho Financial Group's clients experienced a software glitch that collapsed its ATM network and internet banking systems. The result was $1.5 billion in salary payment delays and $9 billion in unprocessed payments. Nine billion. With B.

4. A $2.7 billion US Army cloud computing network failed miserably, leaving troops unable to perform simple operations like sharing data with other users, which, incidentally, is one of the network's main intended functions. You have to wonder how much time and money was ultimately lost — not to mention the number of lives endangered. Not surprisingly, nobody will say; maybe their computers are down.

5. Here's a good one—for those who were able to enjoy the glitch. A Commonwealth Bank ATM network bug caused the machines to dispense large amounts of money to random people. Police actually arrested two people who took the mistakenly spit-out money, saying that it was a crime. No word about the hundreds who took the money and ran—and got away.

Figure 2.1-2: Why Software Reliability is Important

While software reliability impacts many aspects of the product and system total life cycle, it is also subject to influence by the processes and environment in which a product or system is developed. Table 2.1-1 describes some of these influencing factors.

Table 2.1-1: Factors Impacting the Attainment of Reliable Software

Factor	Positive Impact	Negative Impact
Methodologies/tools	Use of structured approaches to design, code, test and maintenance, supported by management	Ad hoc, non-standardized use of methodologies tools, with limited, inconsistent, or non-existent management support
Learning factor	Experience across the organization with structured methodologies and tools	Pockets of experience, or only individual experience, with structured methodologies and tools
Organization	Well-developed organizational guidelines and standards that support an overall business strategy	Ambiguously defined, informal, or no organizational guidelines and standards, with uncertainty as to how reliability fits into the business strategy
Documentation	Well-defined approaches for developing source code and technical references based on comprehensive development plans	Lack of comprehensive development plans and ill-conceived approaches for software documentation (a hacker's paradise)
Environment	Significant understanding of the end-user environment and the ability to successfully model that environment	Limited understanding of the end-user's environment, and, without that understanding, marginal ability to be able to model it

Factor	Positive Impact	Negative Impact
Complexity	Highly structured code and low complexity will result in reliable software. Reliability can still be acceptable as complexity increases, as long as highly structured and disciplined processes are adhered to.	Low complexity code may have acceptable reliability, but as complexity increases, reliability will quickly become unacceptable in an unstructured or poorly structured development process
Prototyping	Effective prototyping during the concept, requirements and design phase	Inadequately structures and disciplined prototyping, or starting prototyping activities too late in the software life cycle
Requirements traceability	Ability to effectively translate and trace requirements during development, based on thorough understanding of customer needs and expectations	Insufficient understanding of customer needs and expectations, resulting in inability to successfully translate and trace requirements to performance
Test methodology	Well-planned methodology that tests and verifies the overall software system reliability based on a comprehensive strategy that considers the entire software life cycle	Inadequate or poorly timed testing that does not adequately precipitate and remove defects from the software before it is delivered to the customer
Maintenance	An approach to maintenance that stresses quick "repair", but not at the expense of disciplined coding, development, documentation and maintenance principals. Maintenance process minimizes introduction of new defects.	Stresses quick "repair", but not much else. Any structure that was inherent in the original software design quickly becomes diluted. Multiple latent errors and downstream effects may be introduced as a result.
Schedule	Entry/exit milestones for objectives are understood and adhered to, and the resources required to meet those objectives on time and within budget planned for	Overall schedule pressures override entry/exit criteria. Resources disproportionately applied to firefighting to solve problems.
Language	Disciplined use of language, whether lower- or higher-order, with proper focus on sound programming and documentation principles	Higher-order languages may inherently provide higher reliability, but undisciplined development will not allow attainment of optimal reliability
Similar software	Characteristics and functionality are understood. Sufficient data exists to support statistically significant modeling and effective interpretation and application of results	Characteristics/functionality not well understood. Supporting data is insufficient, non-existent, or poor quality. Modeling with existing data results in poor models and misleading results.
Qualitative	Maintainability, reusability, safety, fault tolerance, fault containment, security, accuracy, portability, flexibility, performance and user friendliness attributes of the software system are thoroughly understood and can be addressed to the satisfaction of the customer	Limited/no understanding of some, or many, of the qualitative attributes of the software system and how they relate to the internal software development process and/or the explicit/implied needs of the customer

The basic promise of software reliability engineering is to offer a standard, proven best practice that can simultaneously (1) ensure that software reliability meets customers' needs, (2) reduce times to market, (3) reduce development costs, (4) improve customer satisfaction/reduce liability risks, and (5) increase the productivity of software developers and testers. The challenge facing a software developer is how to achieve the proper balance among these characteristics. A product of high reliability may take heavy losses if its delivery to market is delayed and is beat out by a competitor. Likewise, low reliability may result in added liability, expensive retrofit costs and a poor reputation among users. While this goal of proper balance can be somewhat elusive, the process requires the up-front determination of quantitative objectives for each of these characteristics and measurement of progress against these objectives as development proceeds. If a quantitative measure for reliability is lacking, reliability will generally suffer when competing against schedule and cost.

Table 2.1-2, adapted from Reference 6, identifies four technical areas that are applicable to achieving reliable software systems.

Table 2.1-2: Lifecycle Techniques for Achieving Reliable Software Systems (adapted from Reference 6)

Technical Area	Description	Technique
Fault Prevention	Avoid, by design, fault occurrences	Strong requirement specifications, early user interaction and requirement refinement, disciplined software design methods, enforced programming principles and environments, and systematic techniques for software reuse

Technical Area	Description	Technique
Fault Removal	Detect, determine the root cause of, and eliminate faults, and verify and validate that the fix was successful	Software testing and software inspection
Fault Tolerance	Ability of the software to perform to the user's requirements in the presence of faults	Prevent dormant faults from becoming active, prevent software errors from propagating, recover software operations from erroneous conditions, tolerate system level faults
Fault/Failure Forecasting	Estimate the presence of faults and the occurrences and consequences of failure	Understand fault/failure relationship and operational environment, develop software reliability models, and measure software reliability and analyze and act on the results

The basic steps associated with the application of a sound software engineering process (Reference 3) are illustrated in Figure 2.1-3.

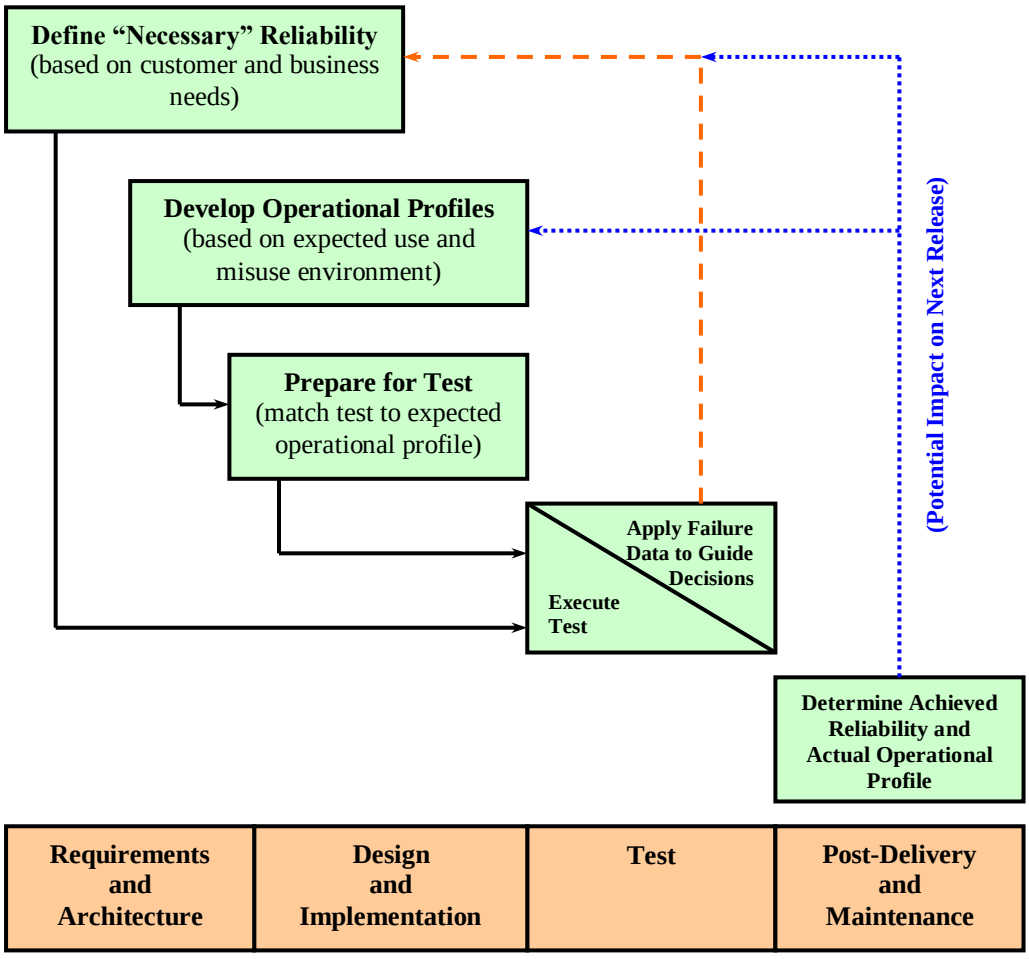


Figure 2.1-3: The Software Reliability Engineering Process (adapted from Ref. 3)

Figure 2.1-4 (Reference 6) provides a similar overview of the software reliability engineering process, highlighting four major areas of the software reliability engineering process approach:

1. Reliability Objective
2. Operational Profile
3. Reliability Modeling and Measurement
4. Reliability Validation

Whereas Figure 2.1-3 places greater emphasis, perhaps, on the operational profile aspects of the process, Figure 2.1-4 provides more detail on the steps associated with software testing, i.e., collecting reliability data; applying available software tools; selecting and using appropriate software reliability models; and validating the results of those models using field data. The overall thrust of these approaches, however, is consistent with the characteristics of a robust software reliability program.

To address security concerns, the operational profile must be greatly expanded and encompass the full range of plausible attack patterns. In addition to the traditional *use cases*, which define desired system interactions by authorized users, *misuse cases* (also called *abuse cases*) must explore ways in which the system might be exploited by malicious agents.

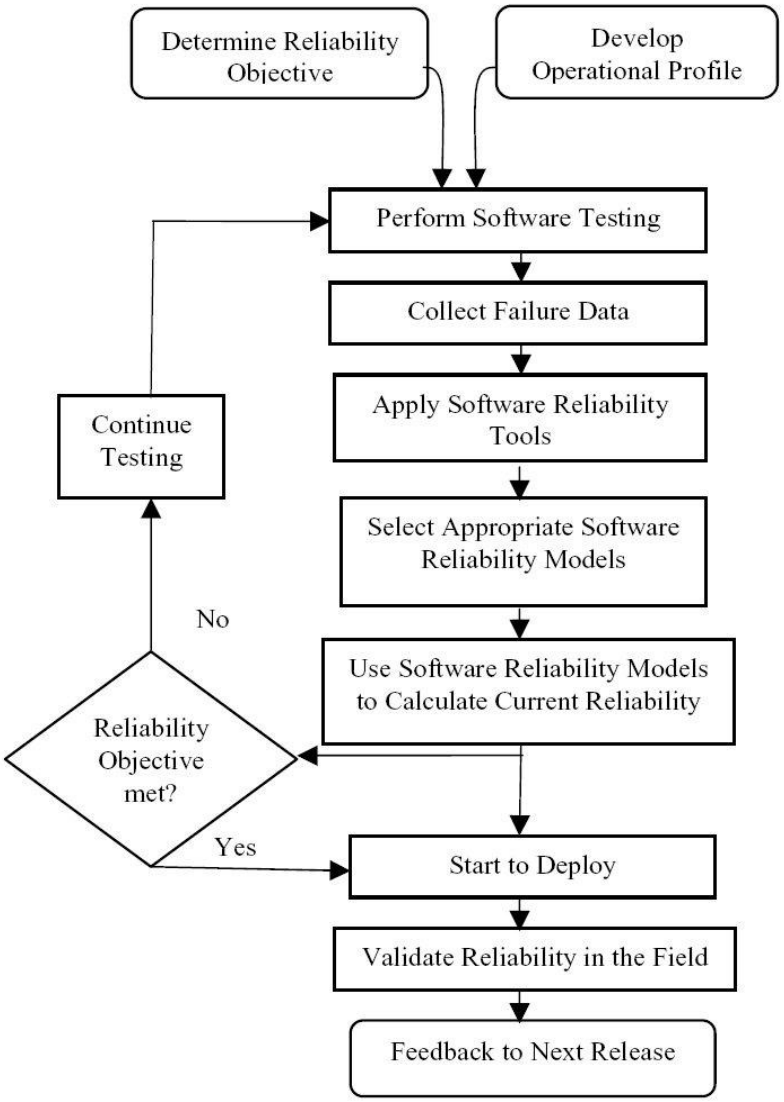


Figure 2.1-4: A Software Reliability Engineering Process Emphasizing Reliability Models and Validation (from Reference 6)

For More Information:

1. Jones, C., "Software Assessments, Benchmarks and Best Practices", Addison-Wesley, 2000, ISBN 0201485427
2. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
3. Musa, J.D., "Software Reliability Engineering: More Reliable Software, Faster Development and Testing", McGraw-Hill, July 1998, ISBN 0079132715
4. Neufelder, A.M., "Ensuring Software Reliability", Marcel Dekker, Inc., 1993, ISBN 0824787625
5. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783
6. Lyu, M.R. "Software Reliability Engineering: A Roadmap", Proceedings of the 29th International Conference on Software Engineering, Minneapolis, May 20-26, 2007, pp. 153-170
7. Musa, J.D., "Software Reliability Engineering: More Reliable Software Faster and Cheaper" (2nd edition), AuthorHouse, 2004, ISBN 1-4184-9387-2 (sc), ISBN 1-4184-9388-0 (dj).

Topic 2.2: Comparison of Hardware and Software Reliability

As discussed in IEEE Std 1633-2008 (Reference 5 in this Section), the creation process of software and hardware products is very similar and can be similarly managed. The literature contains numerous discussions regarding the relationship between hardware and software reliability, discussing their differences and similarities in varying levels of detail. Based on References 2 and 5, Table 2.2-1 provides comparisons of hardware and software reliability fundamentals. Note that throughout this Handbook "fault" and "defect" are used interchangeably, to identify the result of an error (mistake) that injects undesired characteristics in a requirement specification, design element, software code, test document, or other work product.

Table 2.2-1: Comparison of Hardware and Software Reliability Characteristics

Characteristic	Hardware	Software
Failure Cause	Failures can be caused by deficiencies in design, manufacturing, use, and maintenance	Failures are primarily due to design/maintenance faults. Repairs are made by removing the fault or modifying the design to make it robust against the condition(s) that can trigger the failure.
Wear-out	Failures can be due to wear or other energy-related phenomena. Warning may be available before failure occurs through performance degradation.	There is no wear-out phenomenon. Software failures occur without warning. "Old" software code can exhibit an increasing failure rate as functional code upgrades are introduced or due to abrupt changes of its operational usage.
Repairable System Concept	Repairs can be made that may make the equipment more reliable, i.e., preventive maintenance can restore a component within the equipment to like-new condition. Repair generally restores hardware to its previous state (see Reference 5).	Correction of a software fault always changes the software to a new state (see Reference 5). Reliability of a dynamic software system can be enhanced by periodic restarting of the code, re-initializing the operating environment, emptying functional queues and freeing up memory
Time Dependency and Life Cycle	Reliability can be a function of both early life and wear-out. Failure rates can be decreasing, constant, or increasing with respect to operating time.	Reliability is not time dependent. Reliability over time may be impacted by positive or negative reliability growth of the code through detecting, correcting, and introducing, errors.
Time Dependency	Reliability is time related, with failures occurring as a function of operating, non-operating, and/or storage time	Reliability is not time related. Failures occur when a program step or path that contains the fault is executed and triggers a failure. Once fixed, that failure cannot reoccur.
Environmental Factors	Reliability is related to external environmental factors (i.e., temperature, vibration, humidity, etc.)	Reliability is related to the operational usage of the software. The external physical environment has no impact on reliability, except as might result from a change that affects program inputs.
Reliability Prediction	Reliability can be estimated in theory from accurate knowledge of the design, usage and environmental stress factors.	Reliability cannot be estimated from any physical basis. Software reliability prediction is based on available failure data. If failure data is not available (e.g., during the development stages) some "*a priori*" approaches exist based on the software development process used and the complexity of the code.
Redundancy	Mission reliability can be improved by redundancy (at the expense of logistic reliability). The successful use of redundancy presumes ready detection of, isolation from, and switching from failed assets. It also presumes that no common cause failure occurs.	Reliability cannot be improved by redundancy (if parallel paths are identical, each will exhibit the same error) unless each parallel path has different programs written and checked by different teams (N-block redundancy)
Interfaces	Hardware interfaces are visual (e.g., male/female connector interfaces)	Software interfaces (e.g., modules) are conceptual (e.g., parameters or messages)
Failure Rate Drivers	The occurrence of failures in components is somewhat predictable based on device physics and known environmental and operational stresses.	Failure rates are highly dependent on the underlying development process and the complexity of the software. Failures are not usually predictable from analyses of separate statements. Any one statement may be in error. At a system level, interface code and exception handling are the predominant causes.
Standard Components	Hardware uses standard components as the basic design building block (you can buy them from a catalog)	There are few "standardized" parts in software (increased software reuse is addressing this), although there are standardized logic structures
Obsolescence	Parts and manufacturers can become obsolete, although part substitution (form, fit, function) and lifetime buys can extend system life. If not, redesign becomes necessary.	Software becomes obsolete "frequently", as the operational usage of the software changes and as versions are updated for improved functionality/features. System redesign is necessary to extend system life.
Expression of Reliability	Hardware reliability is expressed in wall clock time.	Software reliability may be expressed in execution, elapsed, or calendar time.

As described in Reference 5, despite any differences, hardware and software reliability must be managed as an integrated system attribute. Any perceived differences must be acknowledged and accommodated by the reliability analyses techniques applied.

The concept of both hardware and software failure rate experience over their respective life cycles has historically been presented as the "bathtub curve", presented in Figure 2.2-1. Although illustrated on equal time lines (t_0, t_1, t_2) for comparison purposes, the total life cycle for software is typically significantly shorter than for hardware (software versions will change several times before the hardware will wear out). Table 2.2-2 summarizes what's going on during each phase.

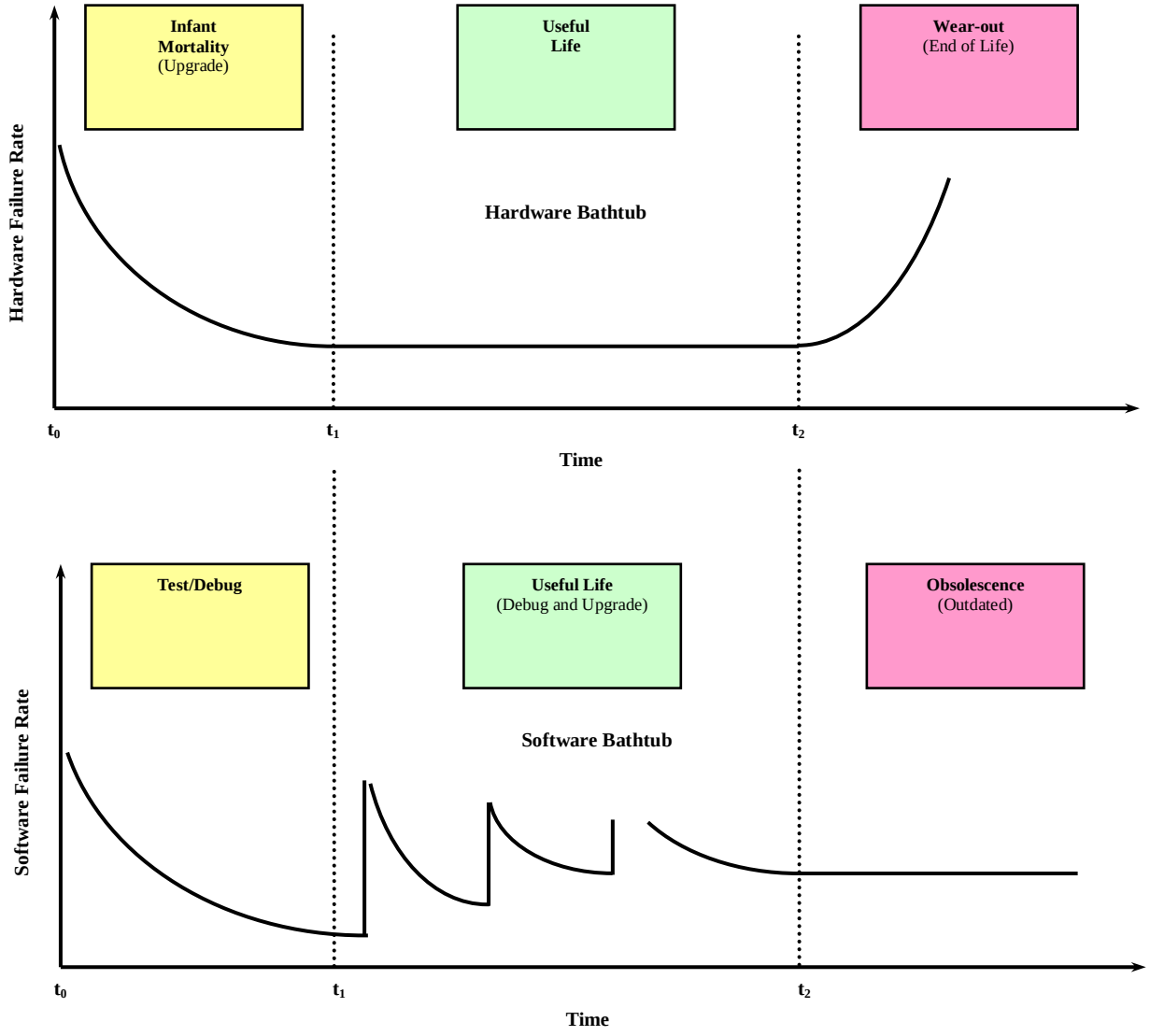


Figure 2.2-1: Comparing the Hardware & Software Bathtubs

Table 2.2-2: What's Going On in the Bathtub

Period	Hardware		Software	
t_0 to t_1	Infant Mortality:	At t_0, the hardware goes into service. Failures result from environmental screening that precipitates out weak parts and manufacturing defects (decreasing failure rate). At t_1, nearly all weak parts and manufacturing defects have been removed from the population.	Test/Debug:	At t_0 testing begins. Coding errors and operational deficiencies are identified and corrected. This differs from hardware in that the development/test time is not counted in hardware failure rate calculations
t_1 to t_2	Useful Life:	Failures occur randomly due to a variety of component failures (constant failure rate). The hardware can be repaired and returned to service with the proper replacement part.	Useful Life:	After software is delivered, failures are found by users, or by continued testing after delivery. These failures are corrected by patches or upgrades (see Reference 6), each of which may introduce new latent failures into the software design. Software may also be upgraded to add new functionality resulting in increased complexity and the possibility of new defects being introduced into the design. Failure rates level off, partly because of the defects found and fixed after the upgrades (see Reference 6).
After t_2	Wear-out:	Equipment starts to exhibit end-of-life failures (increasing failure rate), where it is no longer economical to repair it.	Obsolescence:	In this phase, software is approaching obsolescence, with no motivations for changes or upgrades to the software (see Reference 6). Problems reflect the inability of the software to meet the changing needs of the customer. Although the software functions within spec (not failed), the specs are no longer satisfactory. Problems during this phase can be used as a basis for generating new requirements.

It is fair to observe that the useful life portion of the software bathtub does not necessarily degrade the reliability (i.e., increase the failure rate) of the software. Theoretically, if no changes are made to the software over the time period t_1 to t_2, the software failure rate will remain constant through the remainder of its life cycle at the t_1 failure rate level.

One reason the hardware failure rate decreases during the infant mortality period (t_0 to t_1) is that a conscious effort is made to improve reliability through removal of workmanship defects (process improvements) and environmental screening to eliminate component defects before they are delivered to the customer. The choppy waves during the software useful life phase result from the introduction of latent errors into the design. These newly introduced latent errors may be (1) the unintended result of an attempt to correct previously discovered errors or (2) the result of an upgrade developed to add new functionality to the software. Over time, as defects are discovered and eliminated, the failure rate again decreases until the next change occurs, Whether new defects are introduced by the maintenance process, by feature upgrades, or both, the useful life trend is towards an increased failure rate for mature software. If conscious efforts are made to (1) remove latent defects during the test/debug phase (t_0 to t_1), (2) improve maintenance processes to reduce the number of new defects that might be introduced, and (3) ensure that reliability improvement is a conscious, integrated process with systems engineering, then "choppiness" during the software useful life can lead to a lower failure rate for the mature software.

Software reliability has been slow to evolve as an engineering discipline for several reasons, highlighted in Table 2.2-3. Table 2.2-4 focuses attention on some of the fundamental similarities between software and hardware reliability.

Table 2.2-3: Reasons for the Slow Evolution of Software Reliability

Reason
• Too much focus on hardware and software differences, rather than their similarities • Typical organizational structures do not support effective cross-communication between hardware and software specialists • Hardware-focused reliability engineers have adopted a fairly common set of concepts, terms and definitions, while the software community continues to employ a number of informal or "journalistic" terms (such as "bug" or "glitch"). • Reliability analysis techniques that have been proven effective for hardware only migrated to the software side on a limited basis, even though they can be equally as effective (e.g., FMEA/FMECA, FTA, root cause analysis) • Data collection/analysis and performance of analytical tasks are time- and resource-consuming activities that organizations are generally less willing to invest in. This affects both hardware and software, but hardware reliability is a more mature discipline, i.e., hardware that is designed to be robust is more prone to fail due to deficient processes than failed software components. • Software is intangible. You can't touch it or see it, and its reliability is much more dependent on human action and interaction. It becomes much more difficult to evaluate and fix effectively • The software engineering culture, as a whole, has not developed the discipline required to accept the formality associated with an effective software reliability program (the hardware reliability community has lost a great deal of this discipline as well, through attrition and the changing business environment)

Table 2.2-4: Fundamental Similarities Between Software & Hardware Reliability

Hardware	Software
• Hardware reliability is best achieved by establishing realistic requirements that are aligned with customer expectations, and are explicitly traceable to all tasks performed on a program	• Software reliability is best achieved by establishing realistic requirements that are aligned with customer expectations, and are explicitly traceable to all tasks performed on a program
• Hardware reliability is a function of equipment complexity (number of parts). Generally, the fewer the number of parts, the better the hardware reliability.	• Software reliability is a function of program size and complexity. Generally, the smaller and less complex the software is, the better the software reliability.
• Hardware reliability is a function of applied stresses during normal operation. The better these conditions are understood, the more robust the design can be made to mitigate them.	• Software reliability is also a function of applied stresses during normal operation. The better these conditions are understood, the more robust the design can be made to mitigate them.
• Solid-state electronic devices (microcircuits, transistors/diodes), if fabricated properly, do not exhibit any wear-out failure mechanisms over typical product life spans. Failures that do occur are generally a result of defects that are built in during device fabrication.	• Software, if properly developed, should not exhibit failures during normal operational use. Failures that do occur are generally a result of design faults.
• Higher levels of hardware reliability can be designed in by understanding the predominant failure modes, the environmental stresses (mechanisms) that drive them, and the criticality of each failure using techniques such as failure modes and effects analysis (FMEA), Fault Tree Analysis (FTA), Sneak Circuit Analysis (SCA), and Safety Analysis	• Higher levels of software reliability can be achieved by understanding the predominant failure modes and defect types, their root causes, and the criticality of each failure. Techniques such as failure modes and effects analysis (FMEA), Fault Tree Analysis (FTA), Orthogonal Defect Classification, Operational Profiles, Sneak Analysis, Safety Analysis, and formalized Inspections, to name a few, can be used.
• Hardware reliability can be improved by testing at various levels of hardware indenture, e.g., accelerated testing at the component level, Highly Accelerated Life Test (HALT) at assembly level, and reliability growth testing at the equipment level. At each level of test, emphasis is on verifying and correcting failure modes and causes identified in previous analyses that have not been designed out which may cause premature hardware failure.	• Software reliability can be improved by testing at various software levels, e.g., at the module, CSC, CSCI and system levels. Software reliability can be improved by usage based testing (e.g., via Operational Profiles, Markov models). At each level of test, emphasis is on verifying and correcting failure modes and causes identified in previous analyses that have not been designed out, which may cause premature software failure.
• Hardware reliability is best achieved or improved by continuous improvement in design, manufacturing and maintenance processes, including a closed-loop reporting system for reporting, analyzing, correcting, and verifying the correction of, the root cause of all critical failures during the entire hardware life cycle	• Software reliability is best achieved or improved by continuous improvement in design, development, and maintenance processes, including a closed-loop reporting system for reporting, analyzing, correcting, and verifying the correction of, the root cause of all critical failures during the entire software life cycle

Figure 2.2-2 graphically illustrates the fundamental interrelationships that link software and hardware reliability at the system level over the entire system life cycle.

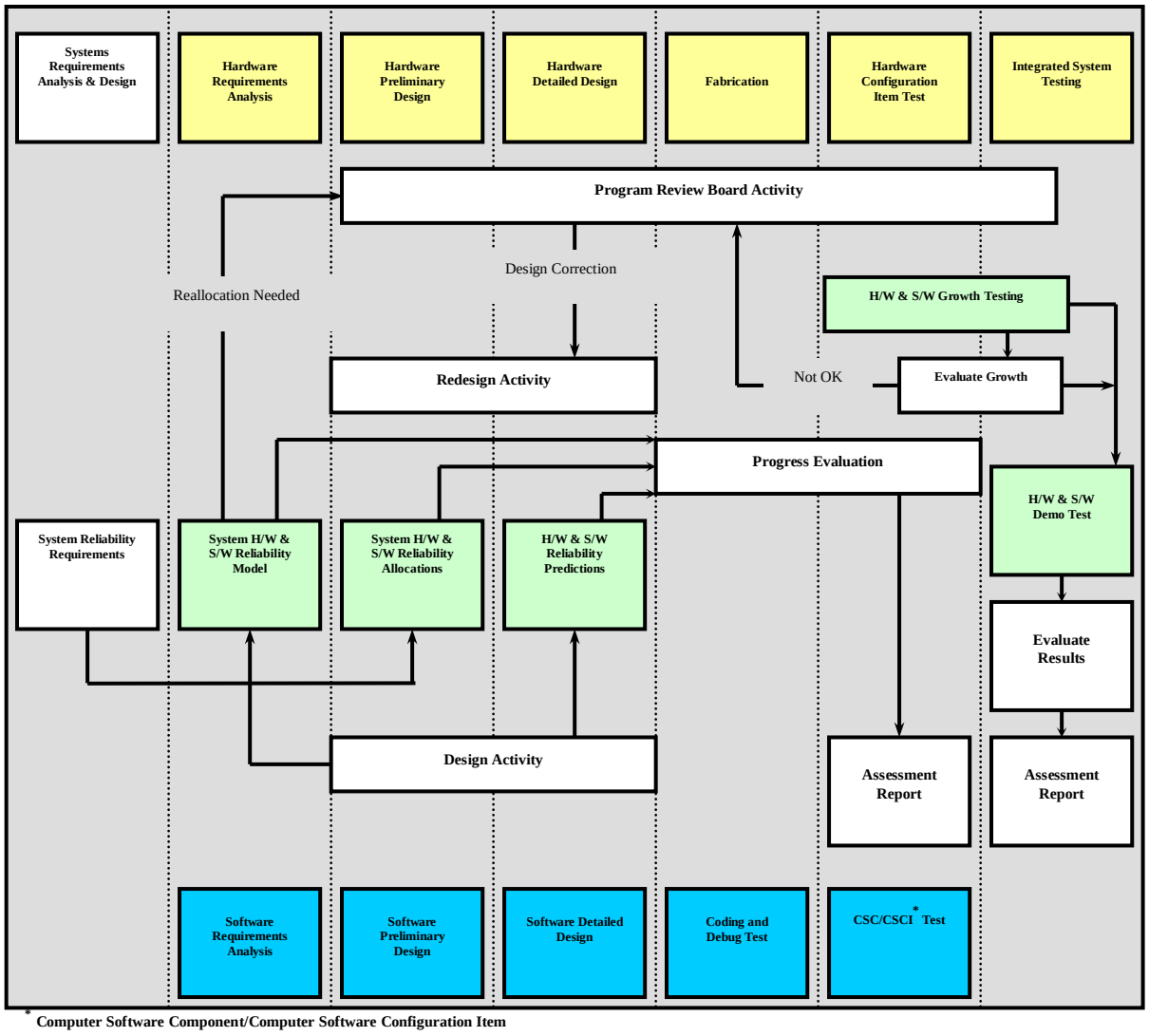


Figure 2.2-2: Relationship of Hardware/Software in the System Life Cycle

For More Information:

1. Keene, S., "Comparing Hardware and Software Reliability", Reliability Review, Vol. 14, No. 4, December 1994
2. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
3. Musa, J.D., "Software Reliability Engineering: More Reliable Software, Faster Development and Testing", McGraw-Hill, July 1998, ISBN 0079132715
4. Neufelder, A.M., "Ensuring Software Reliability", Marcel Dekker, Inc., 1993, ISBN 0824787625
5. IEEE Std 1633™-2008, IEEE Recommended Practice on Software Reliability, IEEE, 27 June 2008
6. Pan, J., "Software Reliability," Carnegie Mellon University, 1999, http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/

Topic 2.3: Software and System Reliability

System reliability is the probability that a system will perform its required functions under stated conditions for a specified period of time. Mission reliability is the probability of mission success. A system may perform many functions or support multiple missions, each of which may have a different reliability.

A system consists of many components, including hardware, software, users, and procedures. The system and mission reliability is a function of the reliability of the supporting components. The system requirements specification is the criterion against which system reliability is measured.

Development of a system, from a reliability perspective, typically involves creating and maintaining a system Reliability Program Plan throughout the system life cycle which states all the reliability tasks that will be performed for that system. It documents what tasks, methods, tools, analyses, and tests are required for a system. The system reliability program plan is developed collaboratively by program management, systems engineers, software engineers, hardware engineers, reliability engineers, and human factors engineers.

One of the first tasks within a reliability program is to specify the system reliability requirements. System reliability requirements are specified as system reliability parameters such as mean time between failure and failure rate. Systems engineers consider various architectures or designs to achieve all systems requirements, and in the analysis of design alternatives estimated system reliability will play a major role in the selection of one architecture over another. Reliability requirements for each alternative, along with design, test and support considerations must be evaluated in the following context:

- Technology maturity
- Life Cycle Costs
- Schedule
- Risk
- Concept of Operations
- Commercial-Off-The-Shelf functionality
- Measurement of effectiveness

During the system design process, system reliability requirements are allocated to hardware and software subsystems.

As with system reliability, software reliability depends on good requirements, design and implementation. Software and system reliability also both rely heavily on disciplined engineering processes to achieve high quality and reliability.

System reliability is a function of software and hardware component/subsystem reliabilities. Reference 2 discusses a simple/basic approach to compute system reliabilities from component reliabilities whose components fail independently of each other:

- Create a success logic expression that shows how system success is related to component success. Block diagrams, Markov state diagrams, or fault trees could be used as an aid in this process.
- All reliabilities must be expressed with respect to a common natural or time unit interval
- For expressions constructed of Boolean AND relationships (i.e., system succeeds only if all N components succeed), the system reliability R is:

$$R = \prod_{k=1}^{N} R_k$$

where R_k is the component k reliability.

- AND relationships can also be expressed in terms of failure intensities as:

$$\lambda = \sum_{k=1}^{N} \lambda_k$$

  where λ_k is component k failure intensity.

- For expressions conducted with Boolean OR expressions (i.e., system succeeds if any component succeeds), analysis can only be performed in terms of reliability. For OR relationships of N components, system reliability R is:

$$R = 1 - \prod_{k=1}^{N} (1 - R_k)$$

It should be emphasized that software differs from hardware in that multiple copies of the same program fail identically and hence represent common mode failures rather than true redundancy. Such copies configured in an OR arrangement do not follow the OR formula. Rather the system reliability would be equal to the common mode reliability.

- Reference 2 provides a shortcut safe approximation for multiplying reliabilities greater than 0.9:
  - Subtract all reliabilities from 1.0
  - Add the corresponding failure probabilities together
  - Subtract the total from 1.0

  For more details, see Chapter 3 of Reference 2.

For More Information

1. IEEE Std 1633™-2008, IEEE Recommended Practice on Software Reliability, IEEE, 27 June 2008
2. Musa, J.D., *Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition).* AuthorHouse. 2004. ISBN 1-4184-9388-0
3. Pham, H., *Systems Software Reliability*, Springer Series in Reliability Engineering, 2006, ISBM 1-8523-3950-0
4. Lakey, P.B., Neufelder, A.M., *System and Software Reliability Assurance Notebook* (Chapter 5.0), downloaded from http://www.cs.colostate.edu/~cs530/rh/index.html

Topic 2.4: Comparison of Software Reliability and Software Security Assurance

Assurance is a term that has been used in many different contexts. It usually refers to a set of activities that, when performed, in turn provide some acceptable measure of confidence.

Quality assurance is defined as "the planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements" (ISO/IEC/IEEE 24765).

Software quality assurance is further defined as "a set of activities that define and assess the adequacy of software processes to provide evidence for a justified statement of confidence that the software processes will produce software products that conform to their established requirements" (IEEE Std 730).

Although the term *software reliability assurance* is not often encountered, confidence in attaining a desired degree of reliability (viewed as a software quality attribute) is part of what is to be achieved through software quality assurance. Conversely, unreliability attributes are to be excluded through aspects of assurance practices that might be identified specifically such as *safety assurance* or *mission assurance* (for example, NASA has an Office of Safety and Mission Assurance: http://www.hq.nasa.gov/office/codeq/).

Confidence in attaining desired security attributes in software-based systems is often referred to as *information assurance or software [security] assurance*. Focusing on the content that is to be secured, *information assurance* is defined as "measures that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation." (Reference 3). Focusing on the securing (software) mechanisms, *software assurance* is seen as the "level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle and that the software functions in the intended manner." (Reference 3).

Software Security Engineering

Most software assurance initiatives are addressing enhancements to best practices in software engineering so as to include addressing security vulnerabilities in software. This enhancement to software engineering is called software security engineering. Many current DoD software systems are interconnected, internet-accessible software-intensive systems susceptible to attack. In this context software assurance and software security engineering address improving software quality to a level that the software system will resist intentional attack as well as unintentional failures.

The Committee on National Security Systems (Reference 3), DoD (Reference 4), and Department of Homeland Security (DHS) (Reference 5) defines software assurance as:

> *"…the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner."*

The Department of Homeland Security further goes on to say that software assurance addresses:

- *"Trustworthiness - No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;*
- *Predictable Execution - Justifiable confidence that software, when executed, functions as intended;*
- *Conformance - Planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements, standards/ procedures.*

Contributing software assurance disciplines, articulated in Bodies of Knowledge and Core Competencies include Software Engineering, Systems Engineering, Information Systems Security Engineering, Information Assurance, Test and Evaluation, Safety, Security, Project Management, and Software Acquisition."

From a reliability perspective, as discussed in Reference 1, whereas software reliability engineering has concerned itself in the past mostly with functional and performance reliability, availability, and dependability, it must now also address software security. DHS says:

"Software Assurance is a strategic initiative of the U.S. Department of Homeland Security (DHS) to promote integrity, security, and reliability in software."

Reference 2 highlights some key aspects of software security engineering:

1. *"Software security is about more than eliminating vulnerabilities and conducting penetration tests. Project managers need to take a systematic approach to incorporate sound software security practices into their development processes. Examples include security requirements elicitation, attack pattern and misuse/abuse case definition, architectural risk analysis, secure coding and code analysis, and risk-based security testing.*
2. *Network security mechanisms and IT infrastructure security services do not sufficiently protect application software from security risks.*
3. *Software security initiatives should follow a risk management approach to identify priorities and what is good enough, understanding that software security risks will change throughout the life cycle. Risk management reviews and actions are conducted during each software development lifecycle (SDLC) phase.*
4. *Developing secure software depends on understanding the operational context in which it will be used. This context includes conducting end-to-end analysis of cross-system work processes, working to contain and recover from failures using lessons learned from business continuity, and exploring failure analysis and mitigation to deal with system and system-of-systems complexity.*
5. *Project managers and software engineers need to think like an attacker in order to address the range of things that software should not do and how software can better resist, tolerate, and recover when under attack. The use of attack patterns and misuse/abuse cases throughout the SDLC encourages this perspective."*

The DHS "Build Security In" website (Reference 6) provides a comprehensive description of best practices relative to software assurance and software security engineering in the following areas:

- Acquisition
- Architectural Risk Analysis
- Assembly, Integration, and Evolution
- Code Analysis
- Deployment and Operations
- Governance and Management
- Incident Management
- Legacy Systems
- Measurement
- Penetration Testing
- Project Management
- Requirements Engineering
- Risk Management
- Security Testing
- System Strategies
- Training and Awareness
- Clear box Testing

NASA Software Assurance

NASA (Reference 7) takes a broader perspective to assurance of mission safety, reliability, and quality in software, beyond just software security, when describing software assurance as a:

> *"planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. It includes the disciplines of Quality Assurance, Quality Engineering, Verification and Validation, Nonconformance Reporting and Corrective Action, Safety Assurance, and Security Assurance and their application during a software life cycle." ... "The application of these disciplines during a software development life cycle is called Software Assurance."*

Table 2.4-1 highlights the major disciplines and activities of software assurance at NASA as discussed in Reference 7.

Table 2.4-1 Software Assurance at NASA

Software Assurance consists of the following Disciplines:	Disciplines of Software Assurance	Role in Software Assurance
Software Quality	• Software Quality Assurance • Software Quality Control • Software Quality Engineering	• Check that standards, processes, and procedures are appropriate for the project • Check that quality attributes, including reliability, are built into the software • Check that the project correctly implements standards, processes, and procedures
Software Safety	• Systematic approach to identifying, analyzing, tracking, mitigating and controlling software hazards and hazardous functions for safer software operation within a system	• Ensure that safety issues are addressed in reviews • Ensure that specific safety analyses and tests are performed • Ensure that requirements pertaining to software's control and monitoring of the safety of the system, personnel, environment are identified and traced throughout the lifecycle
Software Reliability	• Optimize the software by requiring and building in software error prevention, fault detection, isolation, recovery, and/or reduced functionality states • Measure the reliability of products produced	• Ensure that systems are fault tolerant when software fails • Measure and analyze defects in software to find and address possible problem areas within the software
Software Verification and Validation (V&V)	• Ensure that software being developed satisfies functional and other requirements • Ensure that each phase of the lifecycle yields the right products	• Ensure that products of a life cycle phase satisfy the entry conditions of that phase • Confirm that the software will fulfill its intended use
Independent Verification and Validation (IV&V)	• Independently support software risk mitigation • Coordinate with other software assurance disciplines	• Ensure that products that have the highest risk (e.g., safety critical) meet all safety and quality requirements

For More Information:

1. Michael Gegick1, Laurie Williams, Mladen Vouk, "Predictive Models for Identifying Software Components Prone to Failure During Security Attacks;" https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/measurement/1075-BSI.pdf
2. Allen, J.H., Ellison, R.J., Mead, N.R., Barnum, S., McGraw, G., "A Look at 'Software Security Engineering: A Guide for Project Managers,'" CrossTalk, March/April 2010

3. "National Information Assurance Glossary"; CNSS (Committee on National Security Systems) Instruction No. 4009, June 2006, http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf
4. Baldwin, K., Komaroff, M., DoD Software Assurance Initiative, http://proceedings.ndia.org/5871/Komaroff_Baldwin.pdf, accessed on 9 March 2010
5. DHS Software Assurance Community Resources and Information Clearinghouse, https://buildsecurityin.us-cert.gov/swa/, accessed on 9 March 2010
6. DHS Build Security In: https://buildsecurityin.us-cert.gov/bsi/home.html
7. NASA-STD-8739.8, "Software Assurance Standard," 28 July 2004

Topic 2.5: Software Reliability and Risk Assessment

Within the context of systems and software engineering, a probabilistic risk assessment (PRA) is a systematic method to evaluate a wide range of risks associated with system development. PRA starts with establishing what could possibly go wrong that would have a negative impact on a system, and is characterized by establishing and quantifying the magnitude of the potential impact should the undesirable event occur, and the probability that the undesirable event will occur. The total risk is calculated as the sum of the products of the magnitude of the impact multiplied by the probabilities for all undesirable events.

Within systems engineering, PRA is a mitigation strategy to identify and avoid the consequences of a failure from any system failure, regardless of whether it is related to software, hardware, human factors or processes. Knowledge of all aspects of the system design is critical to successfully performing PRA. Systems engineers typically use three methods in performing the PRA:

- Event Sequence Diagrams (ESD). ESD begins with a list of possible failures and analyzes the sequence of events that are likely to happen given that a failure has occurred. Flow chart symbols can be used to enumerate the possible sequence events given a failure.

  Figure 2.5-1 is an example based on Reference 2. This diagram enumerates some of the possible sequences to outcome (end state) for the shutdown of the Space Shuttle Main Engines (SSME) given the first engine shutdown (FES). The first important event to recognize after FES is the choice of whether the redline shutdown software is inhibit (RLI). The up branch indicates "yes" and down means "no". The next important event indicates whether there is a catastrophic failure of at least one of the remaining two engines (CF2) that results in the loss of vehicle (LOV). Assuming "no" on CF2, we are left with the last event indicating whether a second engine is shutdown (SES).

- Event Tree (ET). An event tree is a graphical representation of the logic model that identifies and quantifies the possible outcomes following an initiating event (e.g., a failure). Event tree analysis provides an inductive approach to reliability assessment, as it is constructed using forward logic.

  Figure 2.5-2 is an example event tree based on Reference 4. Event tree analysis, using event tree diagrams, evaluates system response to initiating challenges (e.g., utility system failure) and supports probability assessment of success or failure.

- Fault Trees (FT). Fault trees graphically represent the interaction of failures and other events which may lead to the cause of a failure within a system. Fault tree analysis (FTA) is a failure analysis in which a failure is analyzed using Boolean logic to combine a series of lower-level events. It models failure processes of systems.

  Figure 2.5-3 is an example fault tree from Reference 5. Fault tree analysis is a technique where an undesired state of the system is specified (usually a state that is critical from a safety or reliability standpoint), and the system is then analyzed in the context of its environment and operation to find all realistic ways in which the undesired event (top event) can occur. The fault tree itself is a graphic model of the various parallel and sequential combinations of faults that will result in the occurrence of the predefined undesired event.

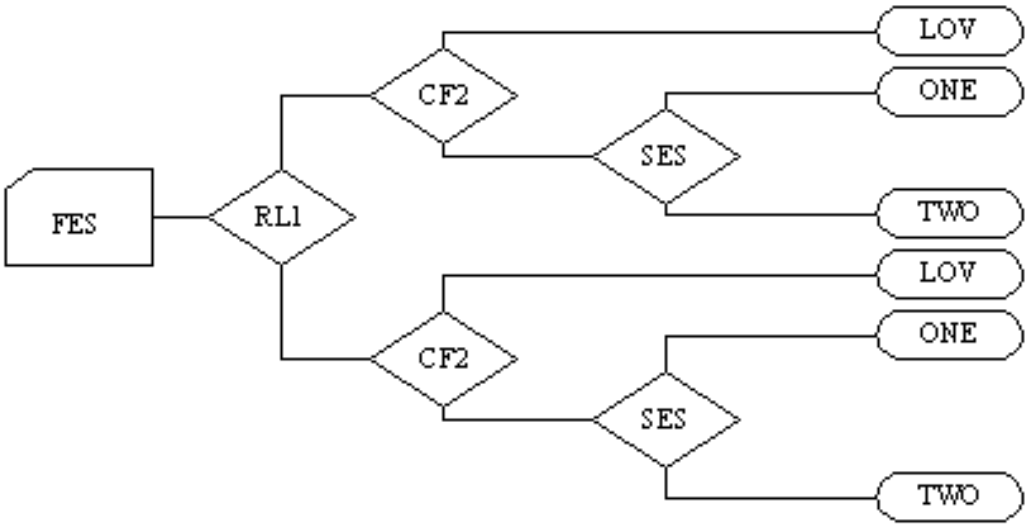


Figure 2.5-1: Example Event Sequence Diagram

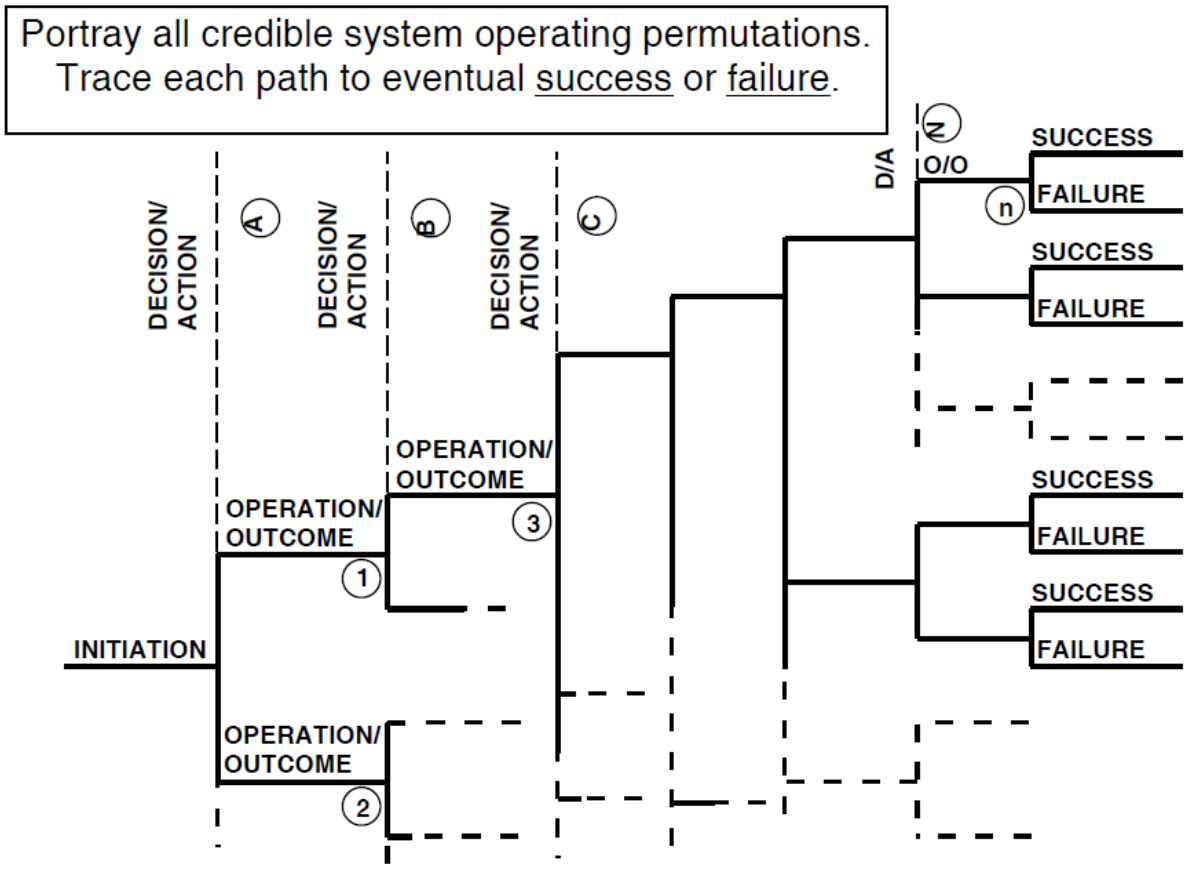


Figure 2.5-2: Example Event Tree

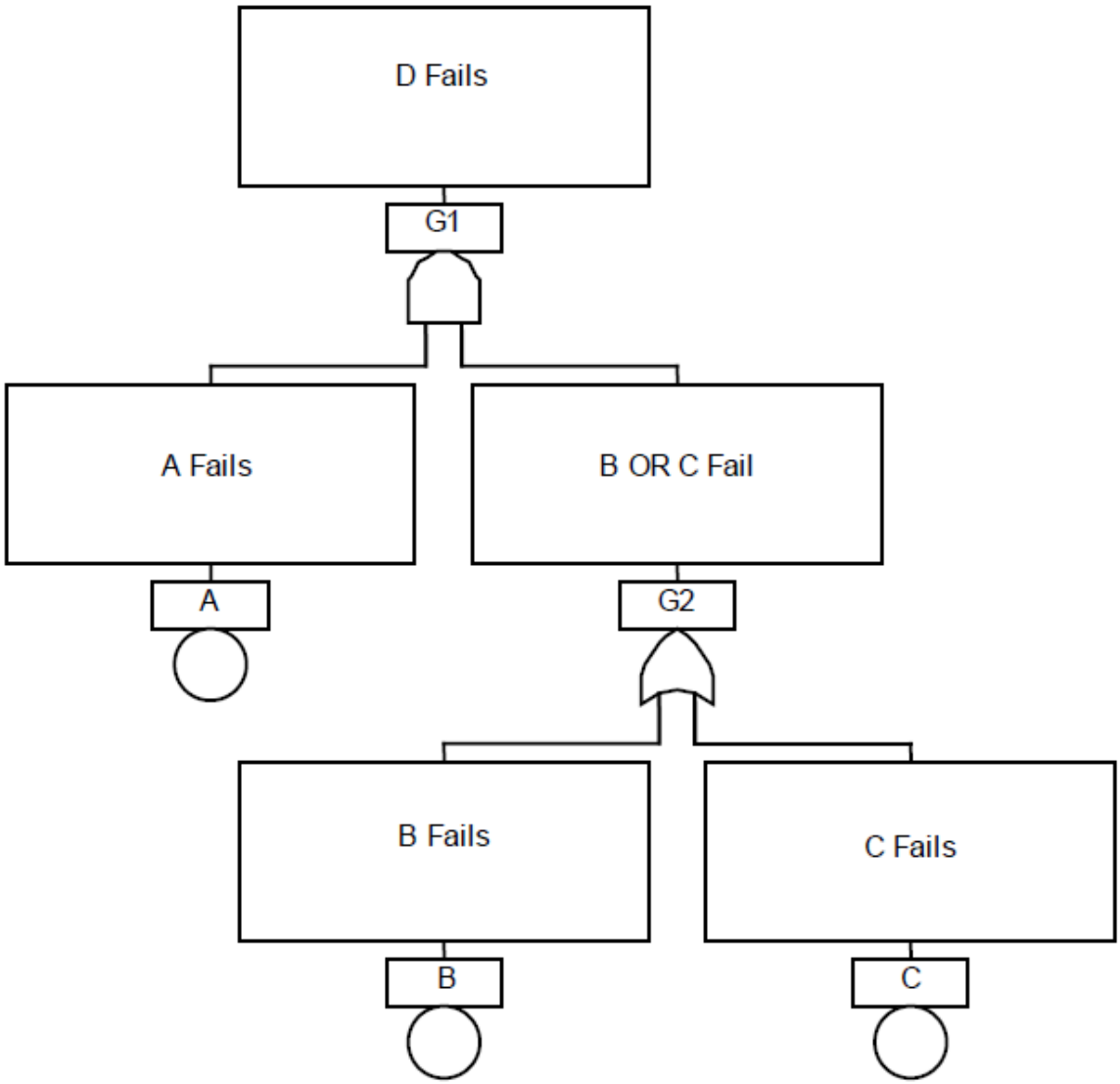


Figure 2.5-3: Example Fault Tree

Current probabilistic risk assessment concentrates on representing the behavior of hardware systems, humans, and their contributions to risk, but typically neglects the contributions of software due to a lack of understanding of software failure phenomena. To include software (i.e., events controlled or supported by software), it is necessary to consider and model the impact of software to reflect the risk. In Reference 3, the authors describe how software contributions to system PRA can be modeled within a classic probabilistic risk assessment using a test-based approach. Based on Reference 3, Table 2.5-1 identifies the major steps and sub-steps required.

Table 2.5-1: Software and Probabilistic Risk Assessment

Steps for Integrating Software into PRA	Activities Within Each Step
(1) Identify events/components controlled/supported by software	• Identify events/components controlled/supported by software in MLD[1], accident scenarios, and fault trees. • For all such events, create/expand contributors to account for software. • Verify that no neglected "events" may now have become possible due to software
(2) Specify the software functions involved in scenarios[2]	• Search requirements for possible functions • To identify the specific functions involved in a scenario, determine the specific input to/output from the software – this will describe one function. • Match the input/output combinations of these functions to the risk model
(3) Model software functions in Event Sequence Diagrams, Event Trees and Fault Trees	• In ESDs and ETs, the modeling of the software function should preserve the function's risk characteristics • In the FT, the top event becomes "the software does not produce the expected output". Caused by: o Failure due to an abnormal input. Probability can be obtained from software testing. o Functional failure under normal input. Probability can be obtained from software testing. o Support failures or failures due to hardware platform failures. Probabilities based on reliability standards.
(4) Software input tree	• Build the input tree for the particular function involved • The input tree is a decomposition of the space of possibilities • The input tree is mostly generic for a function. But may vary due to context. (i.e., probabilities of basic events may vary; certain events may conflict with the rest of the scenario conditions.)
(5) Quantify the Input Tree	(none provided)
(6) Develop and Perform Software Safety Tests	• Build a finite State Machine model of the software by following the software functional decomposition derived from the risk model and the software requirements. • Derive the test profile and output conditions to be quantified from the risk model • Define and run the test cases using the outputs (test scripts) of the finite state machine • Analysis consists of computing the probabilities of the different outcomes based on the test data.

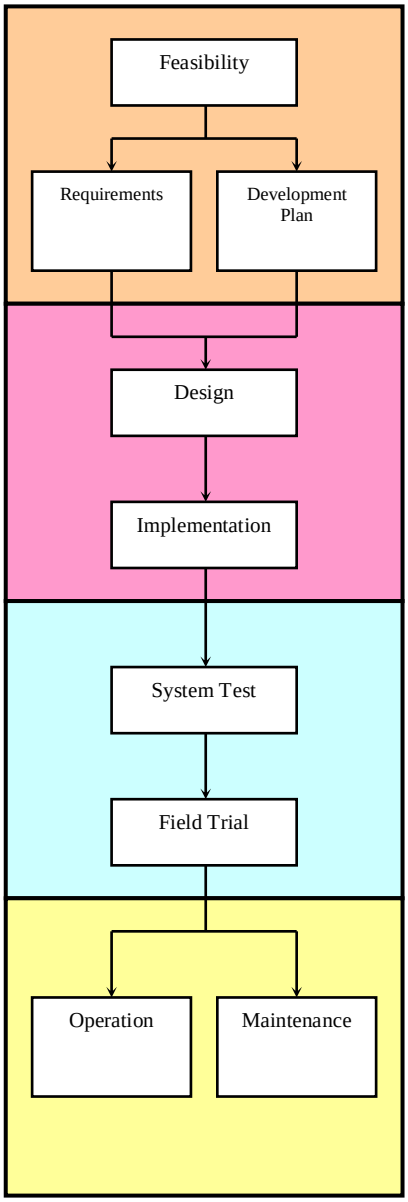
For More Information:

1. Musa, J.D., "Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition)". AuthorHouse. 2004. ISBN 1-4184-9388-0
2. Chhikara, R.S.; Heydorn R.P.; Pitblado, J.S., "Probabilistic Risk Assessment Using Dynamic Event Sequence Diagrams"
3. Li, B., Li, M., Smidts, C., "Integrating Software into PRA: A Test-based Approach
4. Clemens, P.L., "Event Tree Analysis", Sverdrup Corporation, June 1990, available at http://www.fault-tree.net/papers/clemens-event-tree.pdf
5. "Fault Tree Handbook with Aerospace Applications", NASA Office of Safety and Mission Assurance, August 2002

[1] MLD- Master Logic Diagram – used to identify initiating events (See Reference 3)
[2] Not all software functions are involved in fault tree scenarios

Topic 2.6: Reliability Over the System Life Cycle

The process of system reliability engineering is effective only if it is exercised across its entire life cycle. Figure 2.6-1, adapted from Reference 1, provides a conceptual overview of the system reliability process beginning with the "Feasibility and Requirements" life-cycle phase and ending with the "Post-Delivery and Maintenance" life cycle phase. The primary benefits of key activities during each phase of the life cycle are also presented. Table 2.6-1 elaborates on these. It should be understood, however, that in real life there can be considerable overlap and iteration between activities that may very well span across life-cycle phases. As a result, the implementation of system reliability activities should be appropriately tailored for each project, depending on the needs of the customer, the needs of the business, and the structure of the organization. Reviewing the benefits associated with each task will help in deciding which activities will best match these three criteria.



Feasibility and Requirements Phase:

Activity:	Benefit:
• Determine functional profile	1. Speed up time to market (save test time, reduce test cost)
• Define/classify failures	2. Release system when customer needs met, but as early/cheaply as possible
• Identify customer reliability needs	3. Same as 2 above
• Conduct tradeoff studies	4. Increase market share by better match to customer needs
• Set reliability objectives	5. Same as 2 above

Design and Implementation Phase:

Activity:	Benefit:
• Allocate system reliability to lower levels	6. Reduce development time/cost by appropriate reliability focus
• Design to meet reliability objectives	7. Reduce development time/cost with better, more robust design
• Focus resources based on functional profile	8. Speed time to market (guide development priorities, reduce development cost)
• Manage fault introduction/propagation	9. Maximize reliability improvement cost effectiveness
• Measure reliability of acquired hardware, software	10. Reduce reliability, schedule and cost risks due to unknown hardware, software

System Test and Field Trial Phase:

Activity:	Benefit:
• Determine operational profile	11. Same as 1
• Conduct reliability growth testing	12. Know what reliability your customer will experience at different points in time if system were released
• Track testing progress	13. Same as 12
• Project additional testing needed	14. Same as 2
• Certify that reliability objectives are met	15. Same as 2

Post Delivery and Maintenance Phase:

Activity:	Benefit:
• Estimate post-release staff needs	16. Reduce post-release costs with better planning
• Monitor field reliability vs. objectives	17. Maximize likelihood of pleasing customer with reliability
• Track customer satisfaction with reliability	18. Same as 17
• Time new feature introduction by monitoring reliability	19. Ensure that system continues to meet customer reliability needs in the field
• Guide product/process improvement with reliability measures	20. Maximize cost-effectiveness and process improvements

Figure 2.6-1: Reliability Activities Over the System Life Cycle

Table 2.6-1: Profiling the System Reliability Engineering Process

Stage	Activity	Description
Feasibility and Requirements Phase		
Feasibility		• Involves system concept development • Output of this stage defines compatibility between system and market • This stage assesses capability for timely system introduction into market, at a cost, performance and reliability level desired by users relative to relevant competition
	Determine functional profile	• Establish set of product functions (in terms of tasks performed and influencing environmental factors • Account for criticality of functions by weighting them • Use QFD, focus groups or market surveys to identify functions that satisfy user needs
	Define/classify failures	• Define failure from users' perspective, starting with similar systems with the same user base • Distinguish between hardware, software, human and procedural failures • Group failures into a limited number of severity classes, based on effect of failure on users' ability to conduct business, complete mission and/or effect of failure on system safety
	Identify user reliability needs	• Identified at a high level at this stage using a small team (technical, marketing and customer representation) • Establish and assess competitors' reliability capabilities • Determine an approximate acceptable failure intensity for each severity class defined above
Requirements		• Involves preparation of a detailed system requirements specification and Development Plan • Requirements specification expands the needs and high-level features defined during the Feasibility stage, including system reliability, availability, performance and capacity • Development Plan outlines the resources, costs and schedules needed to develop the system • Plan should include adequate resources/schedule for reliability-related activities
	Conduct trade studies	• Set optimal balance of objectives for reliability, cost and delivery date relative to performance of reliability growth testing and current system engineering technology • **Reliability vs. functionality:** Generally, as functionality increases, reliability decreases. Increases in reliability levels equate to increased levels of testing (increased time/cost) • **Reliability vs. cost, delivery date:** Reducing reliability objectives (increased failure rate) reduces reliability growth testing time/cost, but increases field failure costs • **Modeling to support tradeoff studies:** Parameters of some reliability growth models can be predicted from system and development process characteristics. For example, system characteristics for software include newly developed lines of code and lines of reused code. Process characteristics include requirements volatility, design documentation thoroughness and technical personnel experience levels.
	Set reliability objectives	• Separate reliability objectives are established for each failure category, starting at system-level, and then allocated between hardware, software and human factors objectives • Influencing factors include explicitly stated requirements from contractual documentation, reliability performance of/user reaction to previous releases of similar systems, competitors' capabilities, tradeoffs with other characteristics, warranty considerations, reuse of high reliability components, technology capabilities and constraints (e.g., fault-tolerance) • Reliability requirements may strongly influence the architecture and interfaces adopted for a system, and changes in architecture may have a dramatic effect on reliability • System availability (a function of system uptime and downtime) is an important consideration in setting reliability objectives
Design and Implementation Phase		
Design		• Translating a requirements specification into a design of a system • The system architecture is completed during this phase, having evolved through successive iterations • Reliability analysis can assess whether successive iterations will satisfy reliability/availability requirements
	Allocate reliability to components	• Consider alternate ways of dividing the system into components while still meeting overall reliability objectives • Factors to be considered include the nature of the physical system, previously collected data, tracking of critical components, and the resources required for data collection • To determine required component reliabilities, make a trial allocation and calculate the impact on the system reliability. Adjust the allocations to meet the system reliability objective, along with approximate equality of development time, difficulty and risk (tradeoffs)
	Design to meet reliability objectives	• **Plan recovery strategies:** Many system failures may be non-repeatable. Repair of possibly damaged data should be performed prior to any attempt to recreate the failure, and execution should be restarted from a known reference point. Techniques for failure detection, damage confinement and failure recovery should be implemented in the design • **Use redundant system elements:** Redundant system elements for software will improve reliability only if they are not exactly the same (i.e., different programmers develop them independently). Redundancy is generally appropriate only to meet ultra-reliable system or mission requirements (safety-critical). • **Identification of high-risk areas:** Use failure modes and effects analysis (FMEA) and fault tree analysis (FTA) to identify high risk areas in safety-critical system applications
	Focus resources based on functional profile	• Functional profile helps to focus on what is important from the users' viewpoint • Information on frequency of use and criticality of different functions help to weight design alternatives

Table 2.6-1: Profiling the System Reliability Engineering Process (continued)

Stage	Activity	Description
Design and Implementation Phase (continued)		
Implementation		• During the Implementation stage, the design is used to implement the hardware approach and the software code, as well as accounting for human factors issues
	Focus resources based on functional profile	• Functional profile helps to allocate resources during the Implementation stage based on the relative use and criticality of different functions • Functional profile provides guidance for ordering the time periods scheduled for developing system functions (highest use and criticality first)
	Manage fault introduction/propagation	• Using a common approach to development and documentation facilitates good communication to help reduce introduction of faults and design defects into the system • Constructing modular systems using small, simple modules are easier to build and less prone to introduced faults. Modular designs are more maintainable, decreasing the chances that detected faults will be repaired incorrectly. • Reuse of hardware, software and human factors components that were thoroughly tested for a similar operational profile reduces introduction of faults and design defects • Unit tests verify module functionality. Integration tests verify that modules communicate effectively. • Use formal reviews and inspections to verify allocated requirements, design documentation, software code, user manuals, training materials and test documentation • Need to manage the various versions of requirements, documentation and software code and how they are integrated to produce a completed system. There should also be an orderly process for submitting, tracking and completing requested design changes. Reducing the rate of change of requirements generally increases reliability
	Measure reliability of acquired items (NDI/COTS, GOTS, MOTS)	• Need to determine whether acquired items (commercial off-the-shelf, reused hardware or software, etc.) should be certified for a specific application and environment (i.e., a specific environment and/or operational profile) • Certifying the reliability of NDI items can be done via reliability demonstration testing (select test cases at random according to the environment or operational profile and do not fix the underlying faults that cause failures)
System Test and Field Trial Phase		
System Test		• This stage is critical, since it represents the last stage in the development process where corrective action can be taken to improve reliability before release to the first user
	Determine operational profile	• The operational profile is a set of operations and their associated probability of occurrence, where operations are characterized by considering both the tasks performed and environmental factors that influence processing • Two main ways of deriving the operational profile used during testing: One is recording the actual operation of a previous system or an existing similar system. The second is estimating the operational profile, starting from the functional profile developed during the feasibility and requirements phase • It may be necessary to develop different operational profiles for different market segments with different applications, or to fine-tune a special version of the system to meet high reliability requirements for a particular set of functions • For ultra-reliable systems, potential catastrophic failures require extra preventive effort. Ultra-reliable operations should have ultra-reliable objectives, so a separate operational profile may be specifically established and tested for them
	Conduct reliability growth testing	• The goal of reliability growth testing is to attain a level of confidence that a system is being released with a level of reliability that meets user needs • System testers execute test cases in proportion to how often their corresponding operations occur in the field, as characterized by the operational profile • Investing effort in automating the reliability growth test process (test selection, and failure identification and recording) often pays off • Related types of testing are regression testing (ensures that old functions continue to work after repairs or new functions are introduced), feature testing (verifies that system features/functions are present and working properly), and performance and load testing (locates load/stress points at which system objectives are not being met, and certifies that the system satisfies performance objectives such as response time, throughput rates , start-up time and capacity
	Track testing progress and plan additional testing needed	• Failure data is collected and (typically) a software tool is used to track progress and project how much additional test time may be needed • Based on progress, management can make necessary adjustments in resources and schedule as system testing continues
	Certify that reliability objectives are met	• When the current system reliability level reaches its objectives or requirements, the attainment of the reliability objectives or requirements can be certified

Table 2.6-1: Profiling the System Reliability Engineering Process (continued)

Stage	Activity	Description
System Test and Field Trial Phase (continued)		
Field Trial		• When system testing is completed, the system may move to the Field Trial stage (referred to as *beta test* for software) • It is beneficial to have the field trial location use an operational profile that is representative of the conditions of the primary use environment • There should be a field trial plan that documents all failure recording and reporting procedures
	Certify that reliability objectives are met	• Collect failure data from the field site, and use the failure data in conjunction with an automated software tool to measure the reliability of the system in the field • Compare the reliability of the system in the field with the reliability of the system at the end of in-house system testing • When the current system reliability in the field reaches the reliability at the end of system test, the attainment of the reliability objective in the field can be certified • Field trial and system test reliability may differ due to (1) differences between the users' definition of failure and the failure definition applied in testing the system, (2) inaccurate data collection during system test and/or field trials, or (3) differences in the field and test operational profiles, where the test environment may not accurately reflect field conditions
Post Delivery and Maintenance Phase		
Operation and Maintenance		• The system is delivered to, and used by, the user(s) • Maintenance consists of removing all faults associated with any failures that are reported by the user(s)
	Estimate post-release staff needs	• Reliability models can be used to project staff needs following the release of a system • Staff needs may include (1) the user's operations staff to support operation recovery following failure, (2) the supplier's staff to handle failures reported by the user(s), and (3) the supplier's development staff to locate and remove faults/design problems associated with failures reported by the user(s) • When failures are not resolved during operations, reliability models based on constant failure intensities are used to project staff needs for items 1 and 2 • When critical failures need to be resolved, reliability growth models should be used to project item 3
	Monitor field reliability vs. objectives	• Reliability measurements are critical for monitoring the operational reliability of the fielded system • Failure rates may be approximately constant for a given system configuration, but there may be a period of reliability growth (hopefully in a positive direction) just after installation of a new system configuration as a result of field enhancements or repairs • If differences in reliability are observed between the users' environment and what was predicted from system test results, the same possible failure causes listed for the field trial stage should be considered • If differences are observed due to field and test operational mismatch (either in tasks or environmental mismatch), the source of the mismatch must be found and corrected • To perform reliability estimation during operation, collect failure data that is related to the execution time of the software and human performance, as well as hardware performance
	Track user satisfaction	• Select a sample of user sites and survey their level of satisfaction with system reliability • Dissatisfaction may be due to inappropriate objectives/requirements being set, or to other factors related to their use of the system • If there is user dissatisfaction, there should be follow-up to either modify the existing reliability requirements or make necessary changes to the system or field support service process
	Time new feature introduction by monitoring reliability	• Changes to a system that add new functionality also add new latent defects, causing an increase in the failure intensity/rate • If the addition of new features can be segregated from the removal of previous faults, discretion can be used in deciding when the new features should be installed • Failure intensity/rate will increase immediately after new features are added, but periods in which fixes are installed to remove faults will result in decreasing failure intensity. The combination of these phenomena may result in the waves associated with the Useful Life portion of the software reliability bathtub curve • Conflicting demands between users, some wanting new features and others wanting higher reliability of existing features, may require negotiation and establishment of a new reliability objective, where new features are introduced only when the failure intensity/rate falls below a new negotiated reliability objective
	Guide system/process improvement with reliability measures	• Categorize all field failures for analysis by their respective criticality impact on system performance and their rate of occurrence • A root-cause analysis should be performed on each of the selected failures to determine (1) where/why the fault/defect was introduced, (2) why the fault/(defect) escaped detection earlier in the development cycle, and (3) what process changes are needed to reduce the probability that similar faults (defects) will be introduced in the future, or increase the probability that the fault (defect) will be detected at the point of introduction • Implemented process changes should be verified as effective, i.e., proven that the system reliability due to the original fault (defect) has been improved as a result of the corrective action and proven that new faults (defects) have not been introduced as an indirect result of the process change

The technical and administrative functions that should be involved in implementing system reliability, and their involvement over the entire system life cycle, are identified in Table 2.6-2, adopted from Reference 1. Admittedly, only very large organizations would be able to staff a system development project with one individual per job function, and any number of cases could be made for adding or removing certain job functions from certain phases of the life cycle. Depending on the size of the project, the needs of the user, and the resources available to the organization, the job functions involved with system reliability should be suitably tailored to match the overall needs of the business in the market(s) that it serves.

Table 2.6-2: Potential Job Functions Needed to Support System Reliability Engineering

Job Function	System Life Cycle Phase			
	Feasibility and Requirements	**Design and Implementation**	**System Test and Field Trial**	**Post-delivery and Maintenance**
Product Manager	X		X	
Project Manager	X	X	X	
Development Manager		X	X	
Reliability Engineer	X	X	X	X
Systems Analyst				
Systems Engineer	X	X		
Hardware/Software Architect	X	X		
Hardware/Software Designer		X		
Software Programmer		X		
Test Manager	X		X	
Quality Assurance Manager		X	X	X
Test Designer		X	X	
System Tester	X		X	
Installation and Operations Manager	X			X
Users	X			X

One of the most critical aspects of tailoring a system reliability program that simultaneously meets the needs of the user and the needs of the business is to balance the cost of the effort to design, develop, test and redesign the system against the cost to operate and maintain the system once it's delivered to the customer. Decisions related to finding this balance should focus on long-term life cycle cost savings rather than short-term cost savings (shortcuts in design, development, test and redesign), or blindly considering minimization in operating and maintenance costs as the ultimate solution (regardless of how much is invested in design, development, test and redesign to get there). The total life-cycle cost change relationship can be expressed as:

$$\Delta C_{ACQ} + \Delta C_{OM} = \Delta C_{LCC}$$

where,

ΔC_{ACQ} = the change in design, development, test and redesign costs (or acquisition costs), either positive or negative, that correspond to a change in system reliability (increase or decrease)

ΔC_{OM} = the change in operation and maintenance costs (negative or positive) that correspond to a change in system reliability (increase or decrease)

ΔC_{LCC} = the change in total life-cycle cost that corresponds to the changes in the system reliability (positive or negative)

The following general mathematical relationships govern the total life-cycle cost curve:

- As the cost of design, development, test and redesign increases (ΔC_{ACQ}), the cost of operation and maintenance decreases (ΔC_{OM}), and the reliability of the system improves (MTBF increases)

- As the cost of design, development, test and redesign decreases, the cost of operation and maintenance increases, and the reliability of the system declines (MTBF decreases)
- The simultaneous effect of changes in ΔC_{ACQ} and ΔC_{OM} to attain a specific level of system reliability may have a positive or negative effect on the total life-cycle cost, depending on the system MTBF objective
- There is a point where improving system reliability is no longer cost effective (total life-cycle costs begin to increase) - See Figure 2.6-2 for clarification

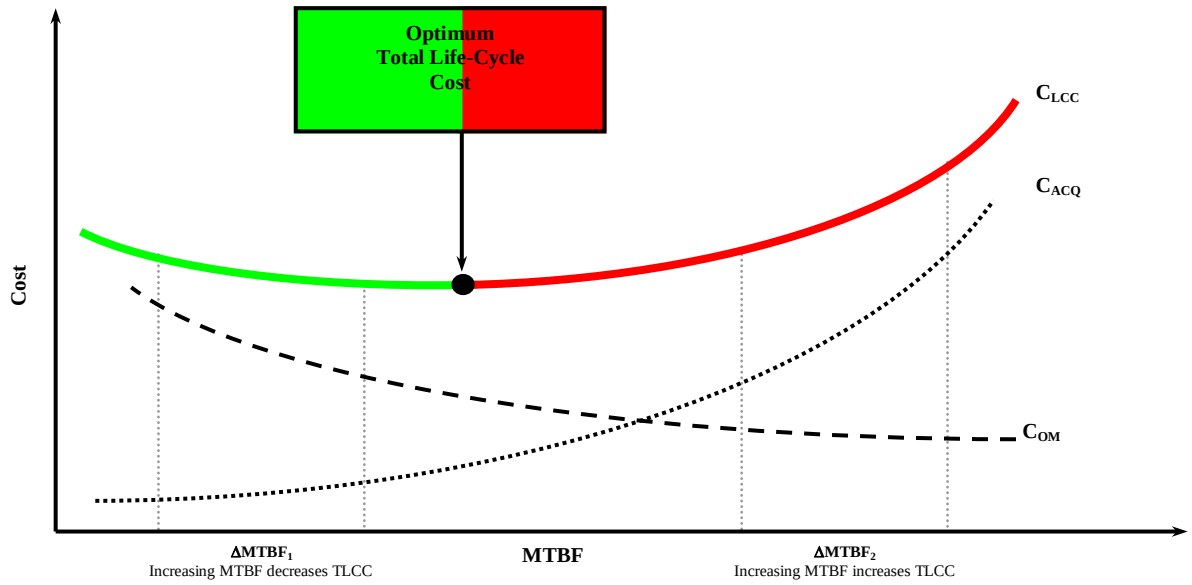


Figure 2.6-2: Determining Optimal Total Life-Cycle Cost

For More Information:

1. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
2. Musa, J.D., "Software Reliability Engineering: More Reliable Software, Faster Development and Testing", McGraw-Hill, July 1998, ISBN 0079132715

Topic 2.7: Identification of System Needs and Feasibility Analysis

The system engineering process begins with the identification of a want or need and is based on a real (or perceived) deficiency in current system capabilities. For example, the current system capability may not be adequate in terms of meeting specific performance goals (e.g., reliability), is not available when needed (availability), cannot be properly supported (supportability), or is too costly to operate (affordability). As a result, a new system requirement is defined along with the priority for introduction, the date when the new system capability is required for customer use, and the anticipated resources necessary for acquiring the new system. To ensure a good start, a complete *statement of the need* should be presented in specific qualitative and quantitative terms, in enough detail to justify progressing to the next step.

Defining the need can be the most difficult part of the systems engineering process. A complete description of the need, expressed in quantitative performance parameters where possible, is essential. It is at this point that the basic requirements for system reliability are first identified. The question is -- *what type of a reliability program is needed?* Although highly "conceptual" at this point, one needs to start answering this question.

The *"needs" analysis*:

> The basic primary and secondary functions that the system must perform are identified, along with the geographical location(s) where these functions are to be performed and the anticipated period of performance. There may be a number of different *technological* approaches considered to be feasible in solving the "need" (i.e., correcting the system deficiency).

The *feasibility analysis:*

> Accomplished with the objective of evaluating different technological approaches that may be considered to meet the specified functional requirements. For instance, in the design of a network, how much bandwidth is needed? In the design of a communication system, should one incorporate a fiber-optics, cellular, or a conventional hard-wired approach? In designing an aircraft, to what extent should composite materials be incorporated?

It is necessary to (1) identify the various possible design approaches that may meet the requirements; (2) evaluate the most likely candidates in terms of performance, effectiveness, life-cycle economic criteria, etc., and (3) recommend a preferred approach. There may be many possible alternatives. The objective, however, is to select the *technical* approach consistent with the state-of-the-art and available resources (time, money, etc.), and NOT to select specific hardware, software, and related system components.

It is in the early stages of the system life cycle when critical decisions will be made. The feasibility analysis results will have a major impact on the overall characteristics of the system. Selection of a specific technology approach has significant reliability and maintainability implications, may significantly affect requirements for spare parts and test equipment, may impact transportation and handling requirements, and will certainly affect the total life-cycle cost (TLCC). Thus, it is at this stage when the overall requirements for *reliability* must be initially addressed.

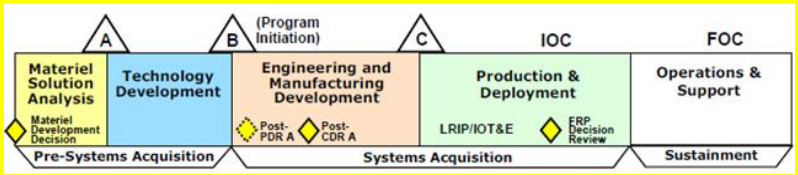
Section 3.0: Testing

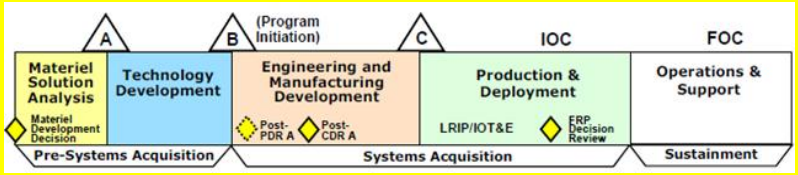
INSIGHT

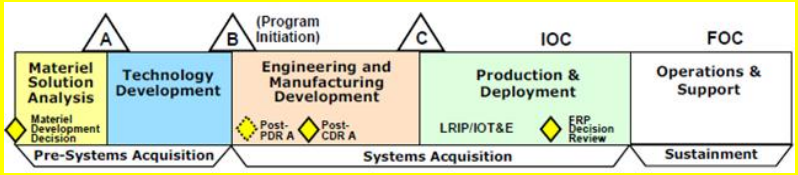
A reliability program can include several forms of testing. While formal reliability qualification tests have become less popular (although they will likely be making a comeback under the new DoD RAM initiatives), they still serve a need for critical missions and unattended operating conditions. The complexity of today's systems has led to resources being shifted toward finding unanticipated design problems (i.e., inherent failure modes) and mitigating them with test-analyze-and-fix or reliability growth testing. These approaches cover not only hardware, but should also consider software and human-machine reliability. Of course, an optimum approach to reliability and reliability growth emphasizes designing it into products through robust Design for Reliability (DFR) processes, rather than depending on testing. Specialized techniques such as accelerated testing and design of experiments (DOE) can effectively be used to conserve precious development resources. Software also benefits from early testing in order to maximize its inherent reliability.

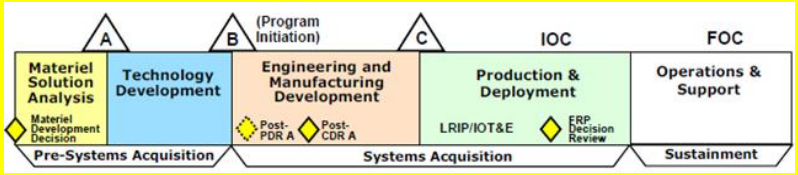
An effective reliability growth test program can help to improve performance reliability and ensure customer and/or end-user satisfaction.

Topic 3.1: Relationship Between Policies/Standards/Guidance and Software Testing

Activity	DoDI 5000.02 Life-Cycle Phase					Comments
						
3.2 System Test Requirements	√	√	√	√		The process of determining the plan for system test and evaluation begins with the initial specifications of system requirements in conceptual design. As specific technical performance measures (TPMs) are defined, it is necessary to determine the methods by which compliance with these parameters will be verified.
3.4 Test Strategies 3.4.1 Software Reliability Test Strategies			√	√	√	Effectiveness of test strategies becomes a combination of the nature of the tests, and the nature of the defects to which the tests are applied. Tests generally fall into the category of "black box" when only a functional understanding exists; "clear box" when detailed understanding of the software exists; and "usage based" when tests are driven by an understanding of how the system will be used
3.4.2 Design of Experiments (DOE)	√	√	√	√	√	DOE allows experimenters to study and quantify the main effects and interactions of factors that influence reliability. DOE statistical methodology for studying the effects of experimental factors on response variables of interest. DOE can be applied to many activities, including design of reliability tests.
3.6 Reliability Growth and Reliability Demonstration/ Qualification Testing			√	√	√	RGT uses generated test and failure data to identify failure modes/mechanisms and find/fix design root failure causes, thereby improving the inherent reliability. Failures are good and should be encouraged. RDT/RQT uses test and failure data to reach statistically valid decisions regarding whether an item has achieved its specified reliability or not. Failures are bad.

Activity	DoDI 5000.02 Life-Cycle Phase					Comments
						
3.6.2 Reliability Growth Testing 3.6.2.3 Software Reliability Growth Testing			√	√	√	Reliability modeling is an essential element of the reliability estimation process. It determines whether a product meets its reliability objective and is ready for release. With reliability growth testing, one or more reliability models are employed to calculate, from failure data collected during system testing, various estimates of a product's reliability as a function of test time. Formal reliability growth testing for software is performed to measure the current reliability, identify and eliminate the root cause of software faults and forecast future software reliability.
3.6.3 Reliability Demonstration/ Qualification Testing (RDT/RQT)			√	√	√	RDT/RQT is conducted as part of the system test and evaluation process. The typical objective of RDT/RQT is to determine if the system under test meets the specified reliability requirements. To accomplish this, the system is operated in a specified manner for a designated time period and failures are recorded and evaluated as the test progresses. Acceptance of the system is based on the system demonstrating a minimum acceptable reliability.
3.5 Software Reliability Testing			√	√	√	The effectiveness of software testing methods, whether they are for detection or demonstration, is directly influenced by the characteristics of the software. Software whose characteristics directly relate back to clear, specific requirements is said to be testable, and the ability of the software to be effectively tested is referred to as its "testability".
3.5.2 Software Test Coverage Metrics			√	√	√	Metrics which measure testing progress based the proportion of tests that have been performed; either measured relative to the number of planned tests or relative to the amount of code tested.
3.5.3 Control Flow Testing			√	√	√	Involves testing based on an awareness of the flow of control throughout a software system. This includes testing each decision in a program based on the logic control flow of the program
3.5.4 Loop Testing			√	√	√	Program loops in software has traditionally been a problematic area for defects in many software systems. Loop testing provides focus on the validity of loop structures. Virtually every repetitive process should be exposed to loop testing.
3.5.5 Data Flow Testing			√	√	√	Data-flow testing selects test paths of a program according to the location and definitions and uses of variables within the program. The basic type of defects that can be expected to be found with Data Flow Testing will tend towards data defects such as those associated with initial and default values; duplication and aliases; overloading; wrong item; wrong type; bad pointers; and data-flow anomalies (such as closing a file before opening it).

Activity	DoDI 5000.02 Life-Cycle Phase					Comments
						
3.5.6 Transaction Flow Testing			√	√	√	Transaction flow testing is designed to find defects in transaction type systems, such as an online or web based applications. Transaction flow systems are characterized by data proceeding along an incoming path that converts information coming in from the outside world into a transaction. This transaction is then evaluated, and where it proceeds next is based on the current transaction value. The transaction-flow graph contains both control-flow and data-flow attributes.
3.5.7 Domain Testing			√	√	√	Domain testing represents a formal technique that can be used when a formal specification (typically algebraic in nature) for a piece of software can be specified.
3.5.8 Finite State Testing			√	√	√	State transition diagrams or finite state machines are very useful design and testing techniques for menu driven, real time, and object oriented systems.
3.5.9 Orthogonal Array Testing			√	√	√	Orthogonal array testing is a statistical black-box testing technique that enables the design of a reasonably small set of test cases when the prospect of exhaustive testing becomes impractical or impossible. The purpose of orthogonal array testing is to assist in the selection of appropriate combinations of factors to provide maximum test coverage from using a minimum number of test cases.
3.5.10 Statistical Usage Testing (SUT)			√	√	√	SUT represents the application of statistical sampling theory to software testing and certification of reliability. The basic premise underlying the use of SUT is that the ability to test all possible ways in which software might be used is impractical. In SUT, testers statistically characterize the population of possible software uses, and how the subset of test cases to be applied will be determined. Usage can be characterized as a tress structure or Markov model.
3.5.11 Operational Profile Testing			√	√	√	Operational Profile Testing, like SUT, attempts to test the software system based on a model of actual system usage. It builds on operational profiles, and tests software based on actual operations as built within the system. Using the operational profile to guide testing can help ensure that, if testing terminates due to schedule constraints, the most-used features of the software will have seen the most comprehensive testing and achieved the maximum reliability level that is practical within the necessary time constraints.
3.5.12 Markov Testing			√	√	√	Markov Testing is another statistical testing technique. It is similar to Finite State testing in that states of a system and transitions are defined. What is different is that each transition is also assigned a probability based on anticipated usage of the system.

Activity	DoDI 5000.02 Life-Cycle Phase					Comments
					
3.5.13 Optimal Release Time			√	√	√	A concept that can be used to determine the optimal release time for software based on cost. It is not necessarily limited to decisions regarding test length, but is appropriate for that purpose, assuming that the failure intensity function is decreasing (reasonable if testing is, indeed, identifying and removing defects without introducing new defects at a faster rate than they are removed). .
4.1 Failure Reporting, Analysis and Corrective Action System (FRACAS)			√	√	√	FRACAS functions as a closed-loop coordinated system in the identification and correction of failures related to product/process, and the identification, implementation and verification of a corrective action to preclude recurrence of the failure. As a result, early elimination of failures is a major contributor to reliability growth and continuous process improvement.
4.1.2 Orthogonal Defect Classification (ODC)			√	√	√	ODC is a methodology and framework which can be used as part of a defect prevention and root cause analysis program to classify and tag software defects into predefined defect classes throughout the development and operational lifecycle. ODC then provides techniques for performing measurement and analysis of the data gathered to gain insight and provide feedback to developers and managers on the progress of a project. Managers can then take proactive measures based on what the ODC data is saying.

Topic 3.2: System Test Requirements

The process of determining the need for system test and evaluation begins with the initial specifications of system requirements in conceptual design. As specific *technical performance measures (TPMs)* are defined, it is necessary to determine the methods by which compliance with these parameters will be verified. This entails determining how the pertinent system TPMs will be measured, as well as the resources required. The approach may be the use of simulation and related analytical methods; employing an engineering model for test and evaluation purposes; testing a production model; evaluating an operational configuration in the user's environment; or using a combination of these techniques. It is necessary to review the requirements for the system, determine the methods that can be used in the evaluation (as well as the anticipated effectiveness of those methods), and develop a comprehensive plan for an overall integrated test and evaluation. Figure 3.2-1 provides a picture of the conventional categories of testing as they are applicable in system test and evaluation. Although not explicitly identified, it is obvious that reliability testing, in general, and software reliability testing specifically, are obvious and critical performance attributes that must be considered as part of successful system testing. A proper test and evaluation program entails a process of preparation and a sequential series of individual categories of tests governed by the phases in the system life cycle.

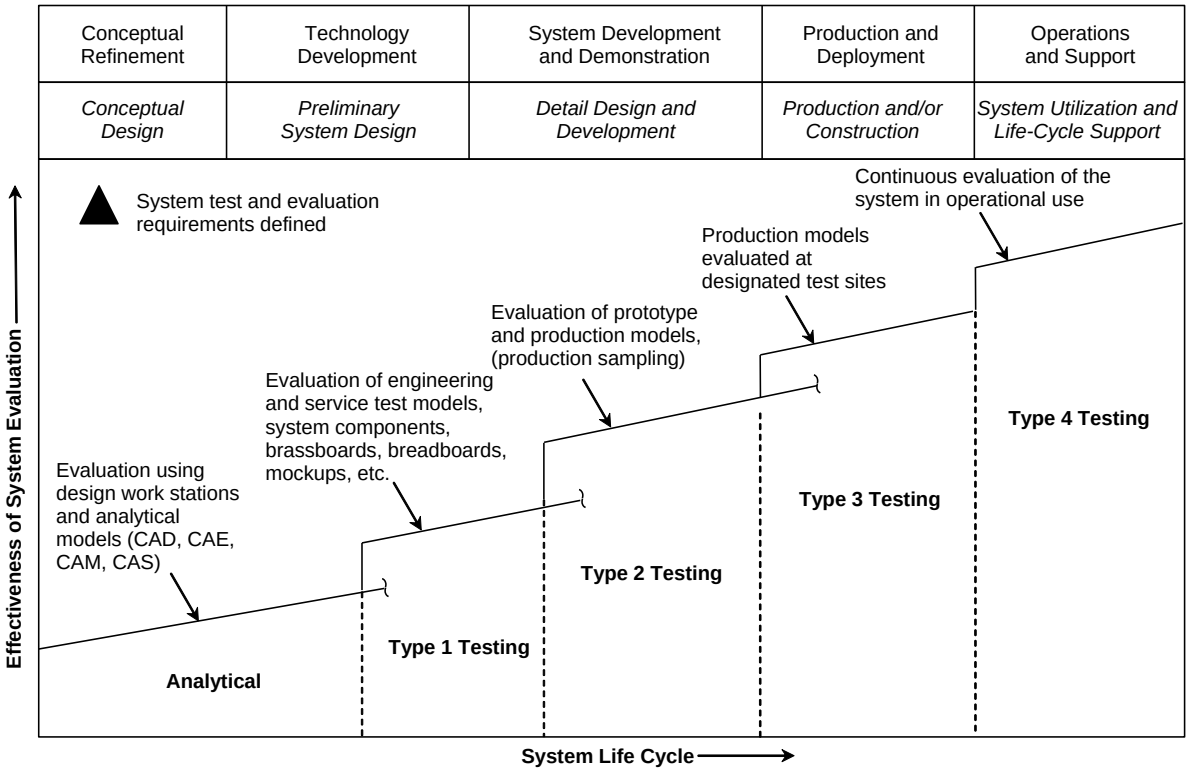


Figure 3.2-1: Stages of System Test and Evaluation During the System Life Cycle

Prior to the start of formal testing an appropriate time period is designated for preparation. During this period, the proper conditions must be established to ensure valid results. These conditions may vary depending upon the category of testing. During the early phases of design and development, as analytical evaluations and Type 1 testing are performed, the extent of test preparation is minimal. Conversely, performance of Type 2 and Type 3 testing, for which the conditions are designed to simulate realistic user operations as much as possible, will likely require extensive preparation. To promote a realistic scenario for test and evaluation, the following factors need to be addressed:

1. *Selection of a Test Item.* The system and its (hardware or software) components selected for testing should represent the most current design or production configuration that incorporates the latest approved engineering changes.
2. *Selection of Test Site*. The system should be tested in the environments that will be representative of the user's environment, e.g., the arctic, tropics or desert; flat or mountainous terrain; airborne or ground environmental or operating profiles. The test site selected should simulate these conditions as much as possible.
3. *Testing Procedures*. The achievement of test objectives usually involves the accomplishment of both operator and maintenance tasks, and the completeness of these tasks should conform to normal procedures (e.g., validated technical manuals). The recommended task sequences must be followed to ensure proper system operation.
4. *Test Personnel*. This group includes (a) the individuals who will actually operate and maintain the system throughout the test, and (b) support engineers, data recorders, analysts, and administrators who provide assistance in conducting the overall test program. Personnel selected for the first category should be representative of user requirements with respect to the recommended quantities, skill levels, and supporting training requirements.
5. *Test and Support Equipment*. The performance of system operational and maintenance tasks may require the use of ground handling equipment, support and test equipment, software, and/or a combination of these elements. Only those items that have been approved for operation should be used.
6. *Supply Support*. This includes all spares, consumables, and supporting inventories that are essential for the completion of system test and evaluation. A realistic configuration projected into a "real-world" environment is highly recommended.
7. *Test Facilities and Resources*. The conduct of system testing may require the use of special facilities, test chambers, capital equipment, environmental controls, special instrumentation, and associated resources; e.g., heat, water, air conditioning, power, telephone. These facilities and resources must be properly identified and scheduled.

Test and evaluation during the system life cycle encompasses the *Analytical* test, *Type 1 Testing*, *Type 2 Testing*, *Type 3 Testing*, and *Type 4 Testing*. It is noted that, within the Department of Defense, Development Test and Evaluation (DT&E) basically equates to the Analytical, Type 1, and Type 2 testing; Operational Test and Evaluation (OT&E) is equivalent to Type 3 and Type 4 testing.

1. **Analytical Test.** The first category is the *analytica*l test, which relates to certain design evaluations that can be conducted early in the system life cycle using computerized techniques including computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided support (CAS), simulation, rapid prototyping, and related approaches. With the availability of an extensive variety of models, three-dimensional databases, etc., design engineers are able to simulate human-machine interactions, equipment packaging schemes, hierarchical structures of systems, and activity/task sequences.
2. **Type 1 Testing.** This testing primarily concerns the evaluation of system components in a laboratory environment using engineering breadboards, bench test models, service test models, rapid prototyping and similar devices. These tests are developmental by nature and are designed primarily for the purpose of verifying performance and physical characteristics of system components. The test models used can operate functionally, but do not represent production equipment or software. Design concepts and technological applications are validated during this initial testing phase, and changes can be initiated with minimum cost impact.
3. **Type 2 Testing.** This testing includes formal tests and demonstrations performed during the latter stages of design and development (System Development and Demonstration Phase) when pre-production prototype equipment and software are available. Prototype equipment is similar to production equipment that will be delivered for operational use, but is not necessarily "qualified" by virtue of successful completion of environmental qualification tests (e.g., temperature cycling, shock and vibration), reliability qualification, maintainability demonstration, and supportability compatibility tests. Type 2 testing primarily refers to

activities associated with the initial qualification of the system. A test program in this area may consist of a series of individual tests, tailored to a specific need, as described in the following:

a. *Environmental Qualification*. Temperature cycling, shock and vibration, humidity, sand and dust, salt spray, acoustic noise, explosion-proofing, electromagnetic interference, etc.

b. *Reliability Screening*. Burn-in, or environmental or highly accelerated stress screening (ESS or HASS)

b. *Reliability Growth and Qualification*. Test, analyze, and fix (TAAF) or more formal reliability growth testing (RGT); reliability qualification or demonstration testing (RQT/RDT); life testing (accelerated or highly accelerated life testing – ALT or HALT).

c. *Maintainability Demonstration*. Verification of maintenance tasks, task times and sequences, maintenance personnel quantities and skill levels, degree of testability and diagnostic provisions, prime equipment-test equipment interfaces, maintenance procedures, and maintenance facilities.

d. *Support Equipment Compatibility*. Verification of the compatibility among the prime equipment, test and support equipment, and ground handling equipment.

e. *Technical Data Verification*. The verification (and validation) of operating procedures, maintenance procedures, and supporting data.

f. *Personnel Test and Evaluation*. Verification to ensure compatibility between the human and equipment, personnel quantities and skill levels required, and training requirements.

g. *Software Compatibility*. Verification that software meets the system requirements, that software and hardware are compatible, and that the appropriate quality provisions have been incorporated. This test includes computer software unit (CSU) and computer software configuration item (CSCI) testing.

h. *Logistics Validation*. Validation of various logistics processes such as procurement, materials handling, transportation, warehousing and distribution, and information.

Another aspect of Type 2 testing is production testing. Although the system design and its components may have successfully passed initial environmental and reliability qualification tests, it is necessary to ensure that the same level of reliability and quality has been maintained throughout the production process. Whether production testing is performed at 100% or on a sample basis will be determined by (1) the number of items being produced, (2) the criticality of reliability or performance in meeting customer or end-user requirements, and (3) recurring satisfactory test results that indicate that 100% testing can be reduced to a suitable sample size. The results are measured and evaluated in terms of whether improvement or degradation has occurred, or whether contractual requirements have been met. A Production Reliability Acceptance Test (PRAT) is an example of a Type 2 test, requiring that a specified level of reliability be demonstrated before the customer will accept delivery of the product or system.

4. **Type 3 Testing.** Type 3 testing entails the accomplishment of formal tests at designated field sites by user personnel over an extended period of time. These tests are typically conducted after initial system qualification and prior to completion of the Production and Deployment Phase of the system life cycle. Operating personnel, operational test and support equipment, operational spares, applicable computer software, and validated operating and maintenance procedures are used. This test marks the first time all elements of the system (i.e., prime equipment, software, and the elements of support) are operated and evaluated on an integrated basis. A series of simulated operational exercises are usually conducted and the system is evaluated in terms of such attributes as performance and effectiveness, as well as the compatibility between the prime mission-oriented segments of the system and the elements of support. Although Type 3 testing does not totally represent a fully operational situation, the tests can be designed to provide an effective approximation.

5. **Type 4 Testing.** This testing is conducted during actual system utilization in the field (Operations and Support Phase) and includes formal tests that are often conducted to acquire specific information relative to a certain area of operation or support. The purpose is to gain further insight about the system in the user environment, or of the user operations in the field. It may be feasible to vary the mission profile or the system utilization rate to determine the impact on the overall system effectiveness, or it may be advisable to

evaluate several alternative maintenance support policies to ascertain if system operational availability can be improved. Type 4 testing should be performed at one or more user operational sites, in a realistic environment, by actual operator and maintenance personnel, and with support through normal logistics and maintenance capability. This is the first time the true capability of the system is assessed.

Topic 3.3: System Operational Requirements

Once a system requirement (i.e., the need) and a technical approach have been identified, it is necessary to expand on the anticipated operational requirements. At this point, the following questions must be asked: what is the specific mission and associated operational scenarios that must be accomplished? Where (geographically) are these scenarios to be accomplished and for how long? What are the anticipated quantities of equipment, software, people, etc., required and where are they to be located? How is the system to be utilized in terms of on-off cycles, hours of operation per designated time period, etc.? What are the expected operational effectiveness goals for the system? What are the expected environmental conditions to which the system will be subjected throughout its operational life?

Table 3.3-1 provides a summary of the key objectives in defining the operational requirements for the system. If one is to design and develop a system to meet a given customer requirement, it is important that the various responsible members of the technical team know the mission objectives and how the system will be utilized in accomplishing these objectives. Of particular interest is the anticipated geographical deployment and type of operational scenarios that are expected to be accomplished. Referring to Figure 3.3-1, a few examples are presented. While one certainly cannot cover all of the future areas of operation, some initial assumptions as to operational scenarios, anticipated utilization, the stresses that the system is expected to see, etc., must be made. The question is -- How can one design without having a pretty fair idea as to how the system is to be utilized? This question is particularly relevant when determining the design requirements for reliability, maintainability, and supportability. Thus, it is appropriate to identify a few of the more rigorous operational profiles and to design with these in mind.

Table 3.3-1: System Operational Requirements

1. Mission definition	Identification of the prime mission of the system and alternative or secondary missions. This may be defined through a series of typical scenarios or operational profiles, and associated system utilization requirements. These scenarios reflect the *dynamics* of the system operating characteristics.
2. Operational deployment (distribution)	Identification of the geographical location(s) where the system and its elements are to be located; i.e., quantity of equipment, software, personnel, facilities, etc., to be distributed and the time duration for distribution.
3. Performance and physical parameters	Definition of the operating characteristics or functions of the system; e.g., speed, acceleration, throughput, accuracy, output, size, weight, process time, etc. These factors must be directly related to the applicable mission/operational scenario(s).
4. Effectiveness factors	Definition of cost-effectiveness, operational availability, dependability, supportability, MTBM, MDT, MLH/ OH, facility utilization, readiness rate, and related requirements. These factors must be directly related to each applicable mission/ operational scenario.
5. Operational life cycle (horizon)	Anticipated time that the system will be in operational use. This represents the baseline for determining the total inventory profile.
6. Operational environment	Definition of the environment in which the system is expected to operate (e.g., temperature, vibration, humidity, arctic or tropics, mountainous or flat terrain, airborne, shipboard, ground fixed or mobile, etc.). This reflects what the system will experience as it accomplishes its mission.

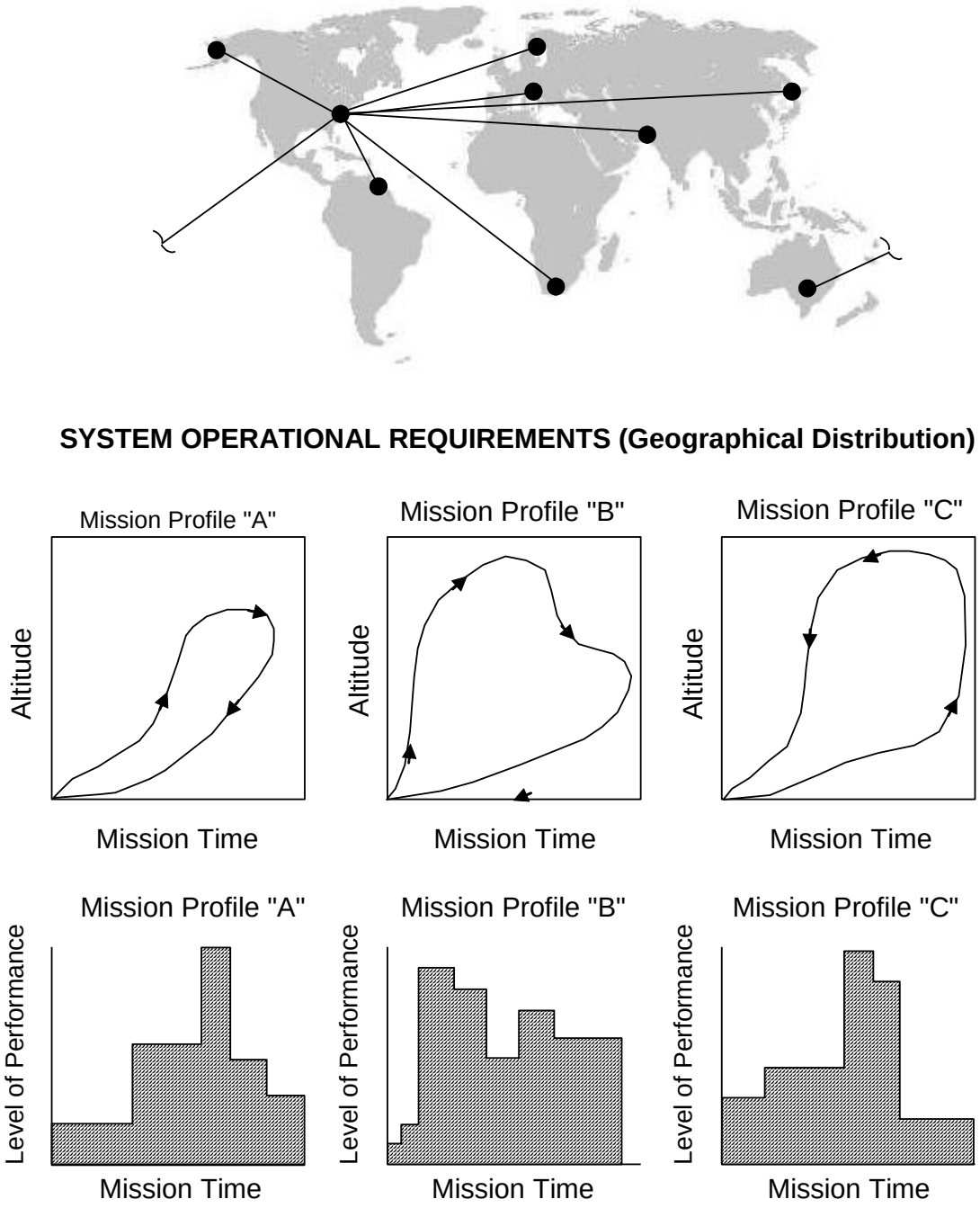


Figure 3.3-1: System Operational Requirements – Example Geographic Distribution and Operational Profiles

Topic 3.3.1: Operational Profiles

System testing of complex software intensive systems is a very complex process, given an almost limitless set of test cases and environmental conditions under which to test. Developing an Operational Profile, which is a probabilistic/quantitative characterization of how a system will be used – or misused -- in the field, provides a cost-effective strategy for developing and executing a test plan which optimizes Reliability Growth during test.

An Operational Profile characterizes the use of a system as a complete list of *operations* performed by a system and an associated *operation probability* that each operation will be invoked. An *operation* is defined as a major system logical task performed for an initiator with control returned to the system when it is complete so that a new operation can be invoked. *Operations* are intended to represent different internal processing from other operations. An operation can be initiated by a user, another system or the system's own controller.

Table 3.3.1-1 provides a tabular example of an Operational Profile (Reference 2) for a hypothetical telephone billing system. In this Operational Profile, operations are classified by type of service (residential, business), type of calling plan (none, national, international), and payment status (paid, delinquent). Billing of (residential, no calling plan, paid) type accounts occur 59.4% of the time; and (business, international, delinquent) type accounts occur 0.03% of the time. With Operational Profile based test planning, tests of the Telephone Billing System would be selected based on their operation probability.

Table 3.3.1-1: Operational Profile for Telephone Billing System (Reference 2)

Operation	Operation Probability
Residential, no calling plan, paid	0.5940
Residential, national calling plan, paid	0.1580
Business, no calling plan, paid	0.1485
Business, national calling plan, paid	0.0396
Residential, international calling plan, paid	0.0396
Business, international calling plan, paid	0.0099
Residential, no calling plan, delinquent	0.0060
Residential, national calling plan, delinquent	0.0016
Business, no calling plan, delinquent	0.0015
Business, national calling plan, delinquent	0.0006
Residential, international calling plan, delinquent	0.0004
Business, international calling plan, delinquent	0.0003

Operational Profiles can also be represented graphically. An example is shown in Figure 3.3.1-1. In this example, when Operation "n" is invoked, Path #2 occurs 20% of the time; when Path #2 is invoked, Path #2.b occurs 80% of the time; etc. Thus, if the probability of Operation "n" occurring is X, then the overall probability of Path #2.b.1 occurring is (0.2 * 0.8 * 0.4) X.

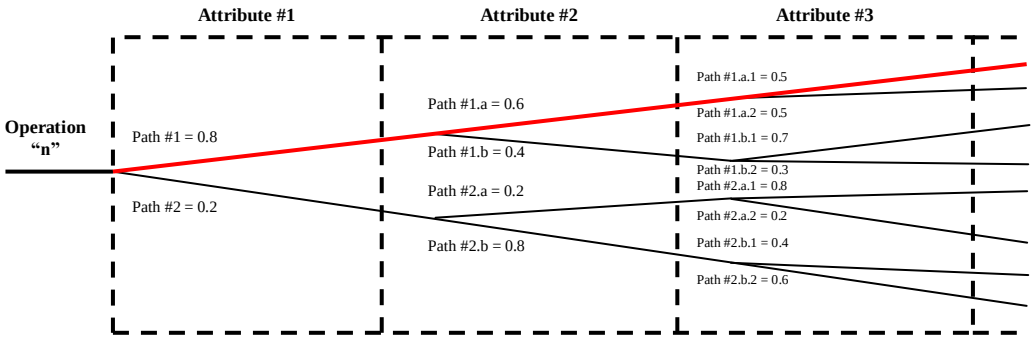


Figure 3.3.1-1: Graphical Representation of an Operational Profile

Developing the Operational Profile

There are five steps in developing an Operational Profile:

1. Identify initiators of operations, including (1) users of the systems, (2) external systems, and (3) operations invoked by the system itself. See Table 3.3.1-2 for details.
2. Create an operations list. See Table 3.3.1-3 for details.
3. Review the operations list. See Table 3.3.1-4 for details.
4. Determine the occurrence rates. See Table 3.3.1-5 for details.
5. Determine the occurrence probabilities. See Table 3.3.1-6 for details.

For Object-Oriented type systems, operations can be derived from *use cases*.

Table 3.3.1-2: Identifying Initiators of Operations

Step	Activities
Identify Users of the System	• Identify expected customer types, based on business case and marketing information. A customer type is a set of customers who acquire your system that have similar business/operational interests. • Identify expected user types for each customer type who directly use the system in the same way. A user is anyone who may initiate operations on the system. Include those that maintain and administer the system. Consider job roles of each user type. • Consolidate user types across customer types
Identify External Systems	• Identify all systems external to the system being developed that interact with the system being developed. Event driven systems often have many external systems that can initiate operations on them. • Consider existing as well as other systems under development or to be developed • Collect technical information on these external systems
Identify Self-Generated Initiators	• Review the system being developed for possible self-generated initiations • Consider administrative functions • Consider maintenance functions

Table 3.3.1-3: Creating the Operations List

Step	Activities
Create a Team of List Operations Experts	• Select personnel familiar with each user type • Select technical personnel familiar with identified external systems • If not the first release of a system, select personnel familiar with previous releases • If possible, select systems engineers and designers • if possible, include typical users
Assemble Needed Technical Data	• Collect and review system requirements documents, use case diagrams, statements of work, work process flow diagrams, draft user manuals, information from previous releases, and prototypes
Identify Operations	• List Operations Experts should o Define operations from the initiator point of view o Identify operations that reinitialize or clean up data (e.g., data reboots) o Identify operations of short duration o Identify operations that perform substantially different processing from the other operations o Identify operations that are testable o Tasks on workflows frequently represent operations o To reduce the number of operations, combine operations that have the most direct input variables in common o For menu-driven systems, "walk the tree" of menus o Maintain traceability between the operational profile and the source material

Table 3.3.1-4: Reviewing the Operations List

Step	Activities
Assemble an Independent Review Team	• Select at least one expert for each initiator (e.g., external system, user type)
Review Operations List for Typical Problems	• Check that: o Operations are of short duration o Each operation has substantially different processing from other operations o Operations are well formed. sending messages and displaying data should be part of the same operation.
Consolidate Operations	The number of operations impacts the number of test cases (at least one test case per operation) which impacts the cost of testing. The anticipated test budget will impact the realistic number of operations and associated tests that can be performed. If operations need to be grouped or consolidated: • Group operations that share the same input variables

Table 3.3.1-5: Obtaining Occurrence Rates.

Step	Activities
	Occurrence Rate = (Number of Occurrences of the Operation) / (Time the Total Set of Operations is Running)
Obtain Field Data	• Obtain existing field data from a previous release or similar system, if available • Obtain "Use" measurements from System Logs, if available • Obtain "business case" type reports which may describe how the system will be used • Develop simulations, if needed. For example, if a system's operational profile us dependent on an external system, simulate the external system
Make Estimates	• If no field data, develop estimates of occurrence rates in conjunction with experienced systems engineers • Involve Users • Apply Delphi Method
Beware of Filler Occurrences	• Filler operations are operations performed by some systems when idle or when there is nothing else to do.
Adjust for Final Occurrence Rates	• Occurrence rates computed from previous releases or other related data needs to be adjusted to account for the new operations, expected changes, environmental changes, and other factors • See Table 3.3.1-7 for an example

Table 3.3.1-6: Determining Occurrence Probabilities

Step	Activities
Estimate Occurrence Probabilities	• Occurrence Probability = (Occurrence Rate of Each Operation) / (Total Operation occurrence Rate) • Table 3.3.1-1 (above) provides Occurrence Probabilities of Table 3.3.1-7

Table 3.3.1-7: Sample Occurrence Rates of Telephone Billing System

Operation	Operation Occurrences
Residential, no calling plan, paid	91,646
Residential, national calling plan, paid	24,377
Business, no calling plan, paid	22,911
Business, national calling plan, paid	6,110
Residential, international calling plan, paid	6,110
Business, international calling plan, paid	1,527
Residential, no calling plan, delinquent	926
Residential, national calling plan, delinquent	247
Business, no calling plan, delinquent	231
Business, national calling plan, delinquent	93
Residential, international calling plan, delinquent	62
Business, international calling plan, delinquent	46
Total Occurrences	154,286

Uses of the Operational Profile

A fully developed Operational Profile provides a wealth of systems usage information to support project planning, development, and testing and test planning in the following ways:

- Use as an aid in developing test plans and testing. See Topic 3.5.11, *Operational Profile Testing*, for details on developing test plans from the Operational Profile. Given that the Operational Plan reflects expected usage of the system, reliability growth is achieved effectively. The most used functions are tested first.
- Use as an aid in allocating development resources. Develop the functions that support the most frequently used operations first.
- Use as an aid in management of releases to customers. Early releases would provide the most frequently used operations.
- Use to allocate system reliability requirements down through the software design hierarchy. Reducing overall project costs by reviewing, during a requirements review, the cost effectiveness of developing software that supports low usage, noncritical operations
- Improving the efficiency of requirements and design reviews by focusing on the most used and most critical operations and functions

Who develops the Operational Profile?

The Operational Profile is usually developed collaboratively by systems engineers, high-level designers, testers and test planners, product planners, marketing personnel, and customers. Although a tool to help system testers to create and execute an effective reliability-growth-based test plan, a level of detailed understanding of how the system is designed (and thus the need for systems engineers and high level designers) is required to understand how the system as built processes operations. Each operational profile is intended to perform different processing from other operational profiles – only an understanding of what is going on within the system provides this insight. Product planners, marketing personnel, and customers provide needed insight on how the system will be used. Starting the process with identification of initiators suggests experts that should be consulted to list operations – each familiar with a particular initiator – and often reveals operations that would otherwise be missed.

How Much Effort is Required to Develop an Operational Profile?

There is limited data available, but as a general rule of thumb, a small system would require about 1-2 weeks to develop. A larger system would require more effort.

When is the Operational Profile Developed?

All five steps for developing the Operational Profile are begun during the requirements phase of the project. The Operational Profile should then be refined in subsequent phases of the project. If a system has a base version and variations, Steps 1-3 are typically the same across the base and variations. Steps 4 and 5 would be unique for the base and each variant. Typically once an Operational Profile is developed for a system, a new release often requires only a review and slight refinement of the results from the previous release.

For More Information:

1. "Test Infrastructure: Domino 8 server reliability in operational profile multi-platform", accessed on December 30, 2009
2. Ozekici, S., Soyer, R., "Stochastics and Statistics Reliability of Software with an Operational Profile," European Journal of Operational Research 149 (2003), pp. 459-474
3. Musa, J.D., "Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition)", AuthorHouse. 2004, ISBN 1-4184-9388-0

Topic 3.4: Test Strategies

Topic 3.4.1: Software Reliability Test Strategies

While defining effective software test cases is important, perhaps more important is the definition and implementation of a strategy by which those test cases should be applied. A small sampling of questions that need to be asked (and answered) as part of the process of developing a software test strategy includes:

- Does a formal strategic test plan need to be developed?
- Should the software program be tested as a whole, or only critical functions?
- Should tests on a system be rerun as new components are added?
- At what point, if any, should the customer become involved in the test?
- How much should the test strategy be driven by product objectives such as safety, reliability, accuracy, usability, or other customer perceptions?

Beizer (Reference 1) defines test strategy as a systematic method that is used to select and/or generate tests that are included in an overall test suite. A strategy should be developed based on a set of rules that address any questions, like those above, by which it can be determined whether a specific test does (or does not) support the strategy. Strategies are only effective if they make visible defects in the software program. Effectiveness of the test strategy, therefore, becomes a combination of the nature of the tests, and the nature of the defects to which the tests are applied. As such, one should be aware of the basic classes and sub-classes of test strategies, as outlined in Tables 3.4.1-1 and 3.4.1-2.

Table 3.4.1-1: Basic Strategies of Software Tests

Test Strategy	Test Class	Synonyms	Comments
Structural	Clear-box	Glass Box White Box Coverage	Uses the control structure of the software program to derive test cases. Test cases can be derived that (1) guarantee all independent paths have been exercised at least once, (2) exercise all logic decisions on their true/false sides, (3) execute all loops at and within their operational boundaries, and (4) exercise internal data structures to ensure their validity. Clear-box testing will detect defects that black-box testing won't (logic defects, incorrect assumptions, design control defects, and typographical defects). Should be performed as early as possible in the software development process.
Behavioral	Black-Box	Functional	Focuses on the functional requirements of the software, enabling the software engineer to derive input condition sets that fully exercise all requirements for a program. Black-box testing attempts to uncover (1) incorrect or missing functions, (2) interface defects, (3) defects in data structure or external database access, (4) behavior or performance defects, and (5) initialization/termination defects. Tends to be applied later in the test process, focusing on the information domain instead of on the program domain.
Hybrid	Combined	None	Combination of structural, behavioral and usage strategies. Unit and low-level components benefit from clear-box tests. Larger components/system testing is appropriate for black-box and usage-based tests. Hybrid strategies prove useful, however, at all levels.
Usage-Based	Statistical Operational Profile Markov	User-Oriented	See Topic 3.5.10 See Topic 3.5.11 See Topic 3.5.12

Table 3.4.1-2: Test Strategy Sub-Classes

Strategy Sub-Class	Clear-box	Black Box	Usage Based
Control-Flow Testing	X	X	
Loop Testing	X	X	
Data Flow Testing	X	X	
Transaction-Flow Testing		X	
Domain Testing		X	
Finite State Testing		X	
Orthogonal Array Testing		X	
Statistical Usage Testing			X
Operational Profile Testing			X
Markov Testing			X
Optimal Release Time	X	X	X
Reliability Growth Testing		X	X
Reliability Demonstration Testing		X	X

- Pressman (Reference 3) discusses control-flow, loop, and data flow testing in the context of clear-box testing
- Beizer (Reference 1) discusses control-flow, loop, and data flow testing in the context of black-box testing

For several items in the table, there is ambiguity as to whether the test is considered better suited for clear-box or black-box testing. If the details of the program coding are known, then clear-box testing can be performed. If that level of detail does not exist and only the basic functionality of the sub-class is known, then black-box testing methods can still be beneficially applied. Remember that clear-box and black-box testing precipitate different types of defects, so there is no obvious advantage of one over the other in this area. The labor intensity and short-term cost (in both dollars and schedule) associated with clear-box testing, however, will typically influence organizations to perform only black-box tests.

Table 3.4.1-3 and Figure 3.4.1-1 provide an overview of how a software test strategy might progress for a large system.

Table 3.4.1-3: Appropriate Test Levels in the Software Strategy

Test Level	Typically Applied Test Classes	Comments
Unit (Component)	Clear-box	Exercise specific module control structure paths to ensure complete coverage and maximize defect detection
Integration	Black-box; limited clear-box	Addresses issues associated with both design verification and product construction
Validation	Black-box; usage-based	Criteria established during requirements analysis must be validated to ensure that all requirements are met
System (Product)	Black-box; usage-based	Software is combined with system hardware, human, and database elements to ensure that overall system performance and functionality are achieved
Regression	Hybrid	Relates to the re-release of a modified software product, where a rerun of the original test suite should be performed

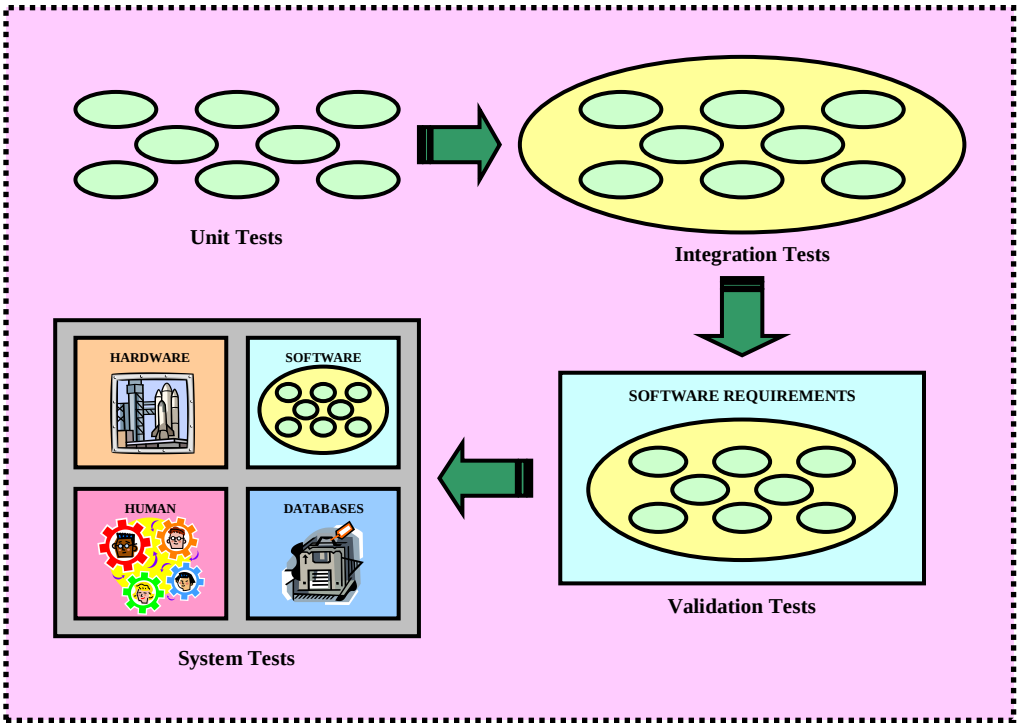


Figure 3.4.1-1: The Test Strategy Implementation Process

Table 3.4.1-4 outlines the general principles for clear-box (or coverage-based) testing. Tables 3.4.1-5 and 3.4.1-6 outline the same for black-box (or functional) and usage-based testing, respectively. Note that the principles for white- and black-box testing are identical, but clear-box tests are primarily focused on design/code, whereas black-box tests are concerned with functionality as defined by the requirements of a specification.

Table 3.4.1-4: General Principles for Clear-box (Coverage) Testing

Principles
• Define the graph elements of the program (nodes, links, weights, entry/exit and loops) • Test the relational properties • Test for node coverage • Test for link coverage (missing/extra/relation, entry/exit/branch) • Test for path and loop coverage (all versus important paths, especially in loops) • Test all weights and properties • "Test": test case, execute, check, follow-up

Table 3.4.1-5: General Principles for Black-Box (Functional) Testing

Principles
• Define the graph elements of the program (nodes, links, weights, entry/exit and loops) • Test the relational properties • Test for node coverage • Test for link coverage (missing/extra/relation, entry/exit/branch) • Test for path and loop coverage (all versus important paths, especially in loops • Test all weights and properties • "Test": test case, execute, check, follow-up

Table 3.4.1-6: General Principles for Usage-Based Testing

Principles
• Capture information in operational profiles - Requirements analysis/gathering - Extrapolation and calibration from existing products - Instrumentation during customer use • Build the usage model - Unconditional probability (Musa) - Conditional probability (Markov chain) - Granularity (functional components) • Execution and result analysis - Select/build evaluation model - Execute evaluation models (preferably in "real-use" applications) - Revise strategy/make decisions based on results

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.4.2: Design of Experiments (DOE)

Design of experiments (DOE) is an efficient, statistical methodology for studying the effects of experimental factors on response variables of interest. The efficiency is primarily achieved through better data collection and utilization, which greatly reduces test times. By applying DOE, the individual effects of a complex system of multiple experimental factors can be studied simultaneously, thereby avoiding the very inefficient "change-one-factor-at-a-time" test approach. DOE techniques can be applied to nearly all facets of product design, process design, and test and evaluation, and are not limited to hardware (See Reference 1 for an application of DOE to evaluate changes to a software development process). It is the intent of this section to give the reader only a brief introduction to design of experiments by providing a single numerical example of what is called a fractional factorial design. Some other competing design strategies, each with their own strengths and weaknesses, include full factorial, Plackett-Burman, Box-Burman, and Taguchi arrays.

Improved levels of reliability can be achieved through the use of design of experiments. DOE allows experimenters to study and quantify the main effects and interactions of factors that influence reliability. These factors may include temperature and voltage, properties such as substrate material and thickness, or outputs from processes such as software development. Once identified, the factors affecting reliability (some of which may be uncontrollable, such as weather) can be systematically and scientifically addressed, ultimately resulting in positive reliability growth.

The generic steps for implementing a robust design approach are listed here. The primary tool is DOE.

- Determine the product feature to be assessed – This feature is referred to as the response of the system
- Determine factors – Factors are the things that can potentially influence the response
- Determine the factor levels – Factor levels are the actual quantitative values of the factors that will be tested in the experiment
- Design the tests – Determine the specific factor-level combinations to be tested, and the order in which they will be tested
- Perform the tests and take measurements in order to generate response data
- Analyze the data to identify the impact that each factor has on the response, and the interactions between each factor
- Determine the optimal settings (or combinations) of the factor levels

The goal of this approach is to determine the factor levels that will result in minimal variability of the product response and maximum probability of the product meeting its requirements.

Figure 3.4.2-1 illustrates the basic concepts associated with a setting up an experimental design.

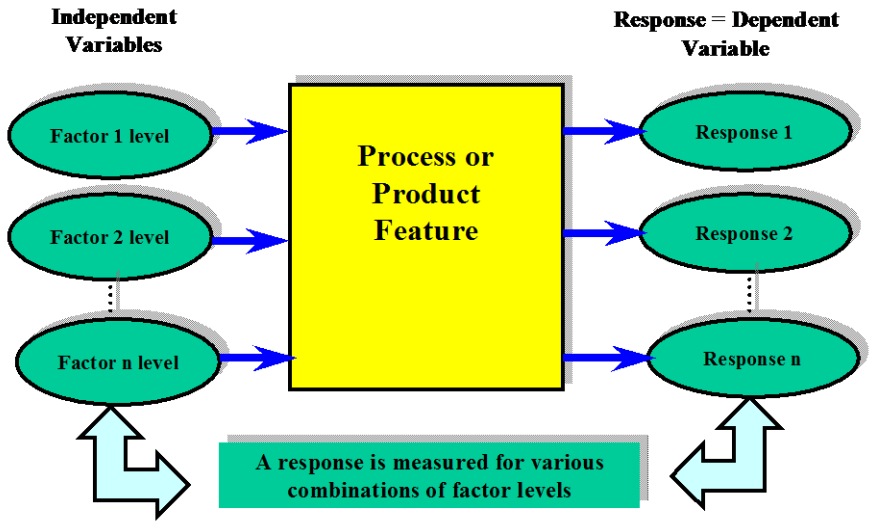


Figure 3.4.2-1: Conceptual View of Design of Experiments

The product or process feature to be assessed can be any quantifiable characteristic of the product that is important to the end user or the producer. It can be related to the performance of the product, or it can be related to the reliability or durability of the product.

A **factor** is any variable that can potentially influence the feature being analyzed. It can be a design attribute, a manufacturing attribute, environmental stress, operational stress, or any other influencing factor. The output of this determination is a list of factors that will be varied in the tests to be performed. A variety of tools can be used to assist in determining the factors that are to be included in the experiments, including (1) Quality Function Deployment (QFD), (2) brainstorming sessions, (3) Ishikawa (fishbone) diagrams, (4) design failure modes and effects analysis (DFMEA), (4) process failure modes and effects analysis (PFMEA) or (5) software failure modes and effects analysis (SFMEA).

After the factors are identified, the next step in the process is to determine the **factor levels** that will be used in the subsequent tests. The simplest and most common approach is the use of two levels, one at the high end of the operating space and one at the low end. However, there are risks associated with using only two levels. The main drawback is that they cannot detect non-linearity in the relationship between the factor and the response. The number of levels for each factor should be chosen, in part, based on knowledge of the manner in which the factor affects the response. For example, if the response under analysis is corrosion, and the relationship between the factor, temperature, and the corrosion rate is expected to be governed by the Arrhenius relationship over the entire operating space, then a two-level temperature test may be appropriate. If, however, it is hypothesized that there is a temperature threshold within the operating space, then more than two levels may be required.

The next step in the process is to **design the experiment** itself. There are many things that will influence the design of the experiment, including sample availability, cost of running the tests, time allotted for the tests, and test item availability. Terminology used in setting up an actual 2-level DOE test matrix is illustrated in Figure 3.4.2-2.

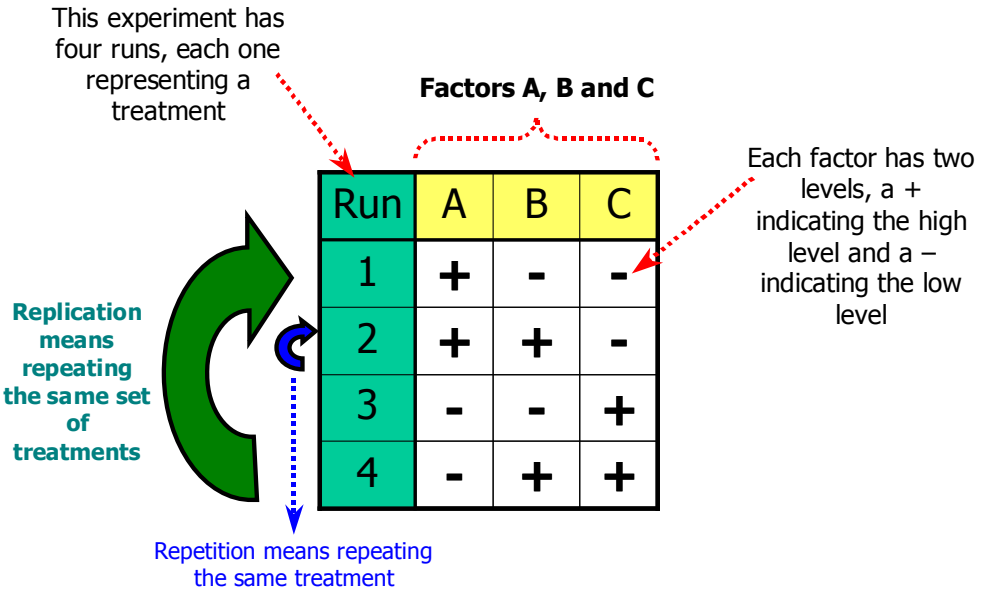


Figure 3.4.2-2: Definitions for the 2-Level DOE Matrix

A full factorial DOE design is the most complete design. It includes runs which represent all possible combinations of factor levels. The primary drawback to the full factorial approach is that it requires many runs. In some cases this is practical, but in most cases, the cost and time required to carry out the experiments are prohibitive. Figure 3.4.2-3 illustrates the matrix for a full factorial design.

Run	A	B	C	R
1	+	+	+	R_1
2	+	+	-	R_2
3	+	-	+	R_3
4	+	-	-	R_4
5	-	+	+	R_5
6	-	+	-	R_6
7	-	-	+	R_7
8	-	-	-	R_8

- **Represents all possible combinations of factor levels**

Number of Runs = y^x

Where:

y = number of levels per factor
x = number of factors

In this example, #Runs = $2^3 = 8$

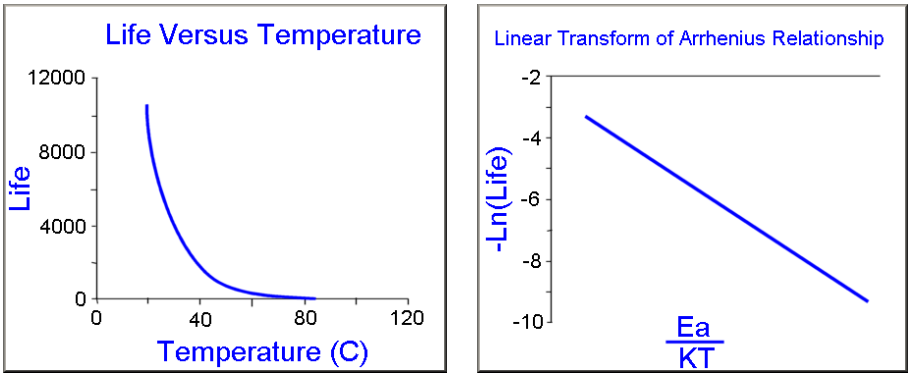
Response

Figure 3.4.2-3: A Full Factorial DOE Matrix

The next step in the process is to **perform the tests**. The test for each run is performed, and the response is measured. All variables that are not factors being addressed in the experiment must be kept as constant as possible. Make sure that all results are fully documented. This also must include any anomalies or potential sources of error that may have occurred. The order of the runs must be kept intact, per the experimental plan. If repetition is used, the same run or treatment is repeated sequentially. If replication is used, then the set of runs to be repeated should be defined in the experimental design.

The simplest way to **analyze the data and the effects of each factor** is to perform an analysis of arithmetic means. In this case, the average value of the response is calculated for each level of each factor. Data analysis techniques

more sophisticated than the analysis of means are often used, and there are many good software tools available to aid in this analysis. However, if a balanced, orthogonal design is used, analysis of means can be very straightforward and effective. In the event that it is known that the response does not behave linearly with the factor level, the response can sometimes be linearized by making the appropriate data transformation. For example, if the response under analysis is corrosion governed by the Arrhenius relationship over the entire operating space, then the response (component or material life in this case) would be exponential with temperature. However, if the transformation shown is applied, the response will be linear. This is especially useful when a goal of the analysis is to determine the activation energy of the corrosion failure mechanism. Figure 3.4.2-4 illustrates this concept.



$$Life = e^{\frac{Ea}{KT}} \quad \text{Linear Transform} \quad Ln(Life) = \frac{Ea}{KT}$$

Ea = Activation energy, K = Boltzman's constant, T = temperature

Figure 3.4.2-4: Linearizing a Non-Linear DOE Response

Now, the **optimal settings of each factor** can be determined.

Everything discussed thus far has assumed that the effects of each of the factors are independent of each other. In practice, there are often interactions between factors that must be accounted for. Examples of interactions are shown in Figure 3.4.2-5. If, for example, the responses for two levels of factor "B" plotted against the two levels of factor "A" are parallel, then this is an indication that there is no interaction between factors. This is shown on the top left plot. In other words, the relative magnitudes of the "B" response are independent of the input levels of "A". If, however, when the same factors are plotted in the same manner results in the plot on the top right, then this is an indication that there is a strong interaction between factors "A" and "B". In this example, the levels of "A" change the entire relationship between the "B" levels and the response. The plot on the bottom indicates that there is a mild interaction between factors.

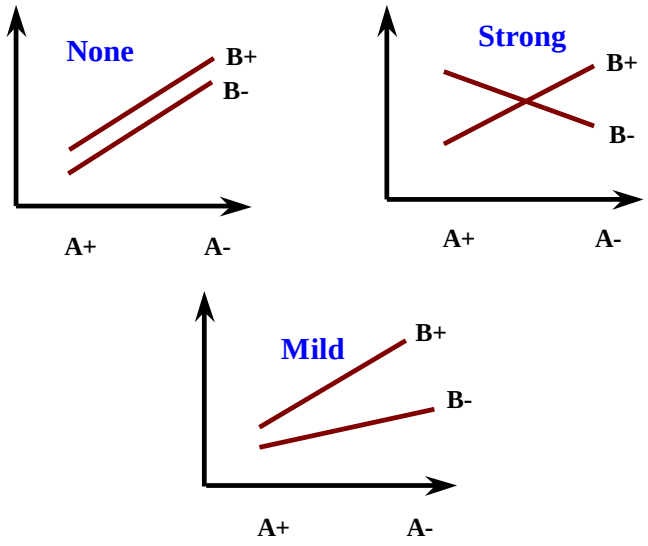


Figure 3.4.2-5: Assessing Interactions in DOE Responses

There are many alternatives to the full factorial approach. "One Factor at a Time" experiments (Figure 3.4.2-6) refer to experiments in which the levels of one factor are varied in successive runs. Each run varies the level of one factor. In this manner, the effects of each factor can be assessed by comparing the response between two successive runs in which the factor was varied. This is generally a brute force way to perform experiments, and is usually very inefficient.

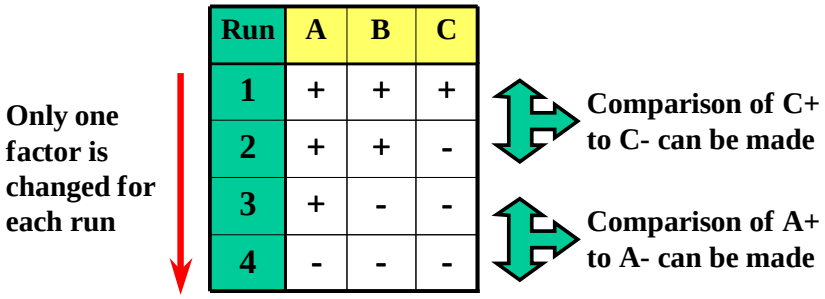


Figure 3.4.2-6: "One Factor at a Time" Experiments

Fractional Factorial Orthogonal Array Experiments can be used when it is impractical to perform a full factorial experiment. Characteristics of orthogonal experiments are:

- They use a fraction of the full factorial combinations
- The treatments are chosen to provide enough information to analyze the effects of a factor using analysis of means
- Orthogonal indicates that the combination of factors are balanced such that the weights of all factors are equal
- Orthogonal also indicates that the effects of the factors can be assessed independently of the others

A full factorial array can be scaled such that the resultant array has the characteristics of orthogonality, as previously described. These are referred to as fractional factorial arrays, since only a fraction of the full factorial runs are required, yet they are still orthogonal. The naming convention for these arrays is given as:

$$L_a(y^x)$$

where,

a = Number of experimental runs
y = Number of levels
x = Number of factors

Figure 3.4.2-7 provides tables of DOE arrays for an L_4 array (where "4" equals the number of runs based on 3 factors, each with 2 unique levels), an L_8 array (8 runs, 7 factors, 2 levels), an L_9 array (9 runs, 4 factors, 3 levels), and an L_{16} array (16 runs, 3 factors, 4 levels).

The following example illustrates the application and usefulness of design of experiments. The example is broken down into a series of steps which reflects the general procedure of DOE discussed above.

Example: Fractional Factorial Design

An integrated circuit manufacturer had determined that a weak bond between a die and an insulated substrate has resulted in many field failures. A designed experiment was conducted to maximize the bonding strength.

Step 1 - Determine Factors: A brainstorming session was conducted which identified four factors believed to affect bonding strength: (1) epoxy type, (2) substrate material, (3) bake time, and (4) substrate thickness.

Step 2 - Select Test Settings: Two test settings ("high" and "low") for each factor were identified. The four factors and their associated high and low settings for the example are shown in Table 3.4.2-1. The selection of high and low settings is arbitrary (e.g., gold eutectic could be "+" and silver could be "-"), but must be consistent.

Table 3.4.2-1: DOE Example Factors and Settings

Factor	Levels	
	Low (-)	High (+)
A. Filled Epoxy Type	Gold	Silver
B. Substrate Material	Alumina	Beryllium Oxide
C. Bake Time (at 90°C)	90 Min	120 Min
D. Substrate Thickness	0.025 in	0.05 in

Step 3 - Set Up An Appropriate Design Matrix: To investigate all possible combinations of four factors, each at two levels, would require 16 (i.e., 2^4) experimental runs. The IC manufacturer decided to use a half replicate fractional factorial with eight runs to conserve time and resources.

The resulting design matrix is shown in Table 3.4.2-2. The "+, -" matrix pattern, defining the factor combinations for the eight runs, was developed utilizing Yates' algorithm (see References 3 and 5). The order of the test runs was randomized to minimize the possibility of outside effects contaminating the data. The matrix is orthogonal, which means that it has the correct balancing properties necessary for each factor's effect to be studied statistically independent from the others. Procedures for setting up orthogonal matrices can be found in any of the references cited.

L_4 (2^3)

	FACTOR		
CASE	A	B	C
1	+	+	+
2	+	-	-
3	-	+	-
4	-	-	+

L_8 (2^7)

	FACTOR						
CASE	A	B	C	D	E	F	G
1	+	+	+	+	+	+	+
2	+	+	+	-	-	-	-
3	+	-	-	+	+	-	-
4	+	-	-	-	-	+	+
5	-	+	-	+	-	+	-
6	-	+	-	-	+	-	+
7	-	-	+	+	-	-	+
8	-	-	+	-	+	+	-

L_{16} (4^3)

	FACTOR		
CASE	A	B	C
1	1	1	1
2	1	2	2
3	1	3	3
4	1	4	4
5	2	1	4
6	2	2	1
7	2	3	2
8	2	4	3
9	3	1	3
10	3	2	4
11	3	3	1
12	3	4	2
13	4	1	2
14	4	2	3
15	4	3	4
16	4	4	1

L_9 (3^4)

	FACTOR			
CASE	A	B	C	D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Figure 3.4.2-7: Examples of DOE Fractional Factorial Arrays

The resulting design matrix is shown in Table 3.4.2-2. The "+, -" matrix pattern, defining the factor combinations for the eight runs, was developed utilizing Yates' algorithm (see References 3 and 5). The order of the test runs was randomized to minimize the possibility of outside effects contaminating the data. The matrix is orthogonal, which means that it has the correct balancing properties necessary for each factor's effect to be studied statistically independent from the others. Procedures for setting up orthogonal matrices can be found in any of the references cited.

Table 3.4.2-2: Orthogonal Design Matrix With Test Results

Treatment Combination	Random Trial Run Order	Factors				Bonding Strength (psi)
		A	B	C	D	y
1	6	-	-	-	-	73
2	5	-	-	+	+	88
3	3	-	+	-	+	81
4	8	-	+	+	-	77
5	4	+	-	-	+	83
6	2	+	-	+	-	81
7	7	+	+	-	-	74
8	1	+	+	+	+	90

$$\text{Mean } \overline{y} = \sum \frac{y_i}{8} = \frac{647}{8} = 80.875$$

Step 4 - Run the Tests: The eight test combinations were run randomly as defined by the table. The run order is determined by a random number table or any other type of random number generator. Resultant bonding strengths from the testing are shown in Table 3.4.2-2.

Step 5 - Analyze the Results: This step involved performing statistical analysis to determine which factors and interactions had a significant effect on the bond strength. Figure 3.4.2-8 shows the reduced set of effects that can be studied as a result of running a fractional replicate. This loss of analysis capability is defined by the aliasing patterns, and is considered the penalty for not checking every possible combination of the factors. Aliases are defined as two or more effects that share the same numerical value. For example, the effect on the bond strength caused by "A or BCD" (column 2) cannot be differentiated between factor A or the interaction of BCD. The assumption is usually made that the effects of higher order interactions (BCD) are negligible and the impact on the response variable was a result of the main factor. Aliasing patterns are unique to each experiment and must be evaluated for reasonableness.

An analysis of variance (ANOVA) was then performed to determine which factors had a significant effect on bonding strength.

Combination Number	A or BCD	B or ACD	C or ABD	D or ABC	AB or CD	AC or BD	BC or AD	Bonding Strength
1	(-)	(-)	(-)	(-)	(+)	(+)	(+)	73
2	(-)	(-)	(+)	(+)	(+)	(-)	(-)	88
3	(-)	(+)	(-)	(+)	(-)	(+)	(-)	81
4	(-)	(+)	(+)	(-)	(-)	(-)	(+)	77
5	(+)	(-)	(-)	(+)	(-)	(-)	(+)	83
6	(+)	(-)	(+)	(-)	(-)	(+)	(-)	81
7	(+)	(+)	(-)	(-)	(+)	(-)	(-)	74
8	(+)	(+)	(+)	(+)	(+)	(+)	(+)	90
Average (+)	82.0	80.50	84.0	85.50	81.250	81.250	80.750	80.875
Average (-)	79.750	81.250	77.750	76.250	80.50	80.50	81.0	
Average(+) - Average(-)	2.250	-0.750	6.250	9.250	0.750	0.750	-0.250	

Figure 3.4.2-8: One Example of a Fractional Factorial Experimental Design

The steps involved in performing an ANOVA for this example (summarized in Table 3.4.2-3) were:

5A. Calculate Sum of Squares: The test data (Figure 3.4.2-8) was used to calculate the sum of squares. The calculation for factor A (filled epoxy type) is illustrated below.

$$\text{Sum of Sq. (Factor A)} = \frac{\text{\# of treatment combinations}}{4} [\text{Avg}(+)-\text{Avg}(-)]^2$$

$$\text{Sum of Sq. (Factor A)} = \frac{8}{4}(2.25)^2 = 10.125$$

5B. Calculate Error: The sum of squares for the error in this case was set equal to the sum of the sum of squares values for the three two-way interactions (i.e., AB or CD, AC or BD, BC or AD). This is known as pooling the error. This error was calculated as: Error = 1.125 + 1.125 + 0.125 = 2.375.

5C. Determine Degrees of Freedom: The degree of freedom of this experiment, "df", is the number of levels of each factor minus one. Degree of freedom is always 1 for factors and interactions for a two level experiment. Degree of freedom for the error (dferr) is equal to 2, since there are 3 interaction degrees of freedom.

5D. Calculate Mean Square: The mean square equals the sum of squares divided by the associated degrees of freedom. Mean square for a two level, single replicate experiment is always equal to the sum of squares for all factors. Mean square for the error is equal to the sum of squares error term divided by 3 (where 3 is the "df" of the error).

5E. Perform F-Ratio Test for Significance: To determine the F-ratio, divide the mean square of the factor by the mean square error. The result is statistically distributed according to the F-distribution, and is compared to the value defining the critical region. $F\{\alpha, df_F, df_{err}\}$ represents the critical value of the distribution and is tabulated in most statistics books. If the F-ratio is greater than the critical value, then the null-hypothesis (the factors studied had no effect on the response) is rejected, and the factor is assumed to have a significant effect on the response variable. Alpha (α) represents the risk of rejecting a true null-hypothesis. For this example, assuming a 10% risk, the critical value was $F\{0.1,1,2\} = 8.53$.

The above formulations are not intended to be used in a cookbook fashion. Proper methods for computing sum of squares, mean square, degrees of freedom, etc., depend on the type of experiment being run and can be found in appropriate Design of Experiments reference books.

Table 3.4.2-3: Results of Analysis of Variance for Example

Source	Sum of Squares	Degrees of Freedom	Mean Square	F Ratio*	Significant Effect
Epoxy Type (A)	10.125	1	10.125	8.52	Yes
Substrate Material (B)	1.125	1	1.125	0.95	No
Bake Time (C)	78.125	1	78.125	65.76	Yes
Substrate Thickness (D)	171.125	1	171.125	144.04	Yes
A*B or C*D	1.125	1	--	--	--
A*C or B*D	1.125	1	--	--	--
B*C or A*D	0.125	1	--	--	--
Error	2.375	2	1.188	--	--

*Example Calculation: F = Mean Square/Error = 10.125/1.188 = 8.52

Step 6 - Calculate Optimum Settings: From the ANOVA, the factors A, C, and D were found to be significant at the 10% level. In order to maximize the bonding strength, the optimum settings were determined by inspecting the following prediction equation:

$$y = (\text{mean bonding strength}) + 1.125A + 3.125C + 4.625D$$

Since A, C, and D are the only significant factors, they are the only ones needed in the prediction equation. Further, because they all have positive coefficients they must be set at high to maximize bonding strength. Factor B, substrate material, did not significantly affect bonding strength, so the choice of material should be based on cost. An economic analysis should always be performed to ensure that all decisions resulting from designed experiments are cost-effective.

Step 7 - Perform Confirmation Run Test: Since there may be important factors not considered, the optimum settings must be verified by test. If a confirmation test supports the DOE results, the job is done. If not, new tests must be planned.

For More Information:

1. Wakeland, W.W., Martin, R.H., Raffo, D., " Using Design of Experiments, Sensitivity Analysis, and Hybrid Simulation to Evaluate Changes to a Software Development Process: A Case Study", Portland State University
2. Barker, T. B., "Quality By Experimental Design," Marcel Dekker Inc., 1985
3. Box, G.E.P., W. G. Hunter, and J. S. Hunter, "Statistics for Experiments," John Wiley & Sons, New York, NY, 1978
4. Davies, O.L., "The Design and Analysis of Industrial Experiments," Hafner Publishing Co.
5. Fisher, R.A., and F. Yates, "Statistical Tables for Biological, Agricultural and Medical Research," (4th. Ed.). Edinburgh and London: Oliver & Boyd, Ltd. 1953
6. Hicks, C.R., "Fundamental Concepts in the Design of Experiments," Holt, Rinehart and Winston, Inc., New York, NY, 1982
7. Schmidt, S. R. and R. G. Launsby, "Understanding Industrial Designed Experiments," Air Academy Press, Colorado Springs, CO, 1989
8. Taguchi, G., "Introduction to Quality Engineering," American Supplier Institute, Inc., Dearborn, MI, 1986

Topic 3.5: Software Reliability Testing

Topic 3.5.1: Overview

The purpose of this topic is to recognize that testing currently plays a critical role in the success of both large and small software projects, and will continue to do so in the foreseeable future (unless someone comes up with a fool-proof, repeatable process for developing and integrating software that possesses perfect reliability (zero defects), in which case testing of software will become obsolete).

Table 3.5.1-1 identifies several reasons why software testing needs to be performed. In all cases except one (reliability demonstration testing), the success of a test is measured by the number of defects that it detects (assuming that they are ultimately corrected, resulting in positive reliability growth), not by completion without failures.

Table 3.5.1-1: Reasons to Test Software

Reason	Comments
Detect, expose and correct defects	Defects can be in code, requirements and/or design. Gives programmers information they can use to prevent future defects.
Demonstrate that requirements have been satisfied	The rationale for any test should be directly traceable to a customer requirement (whether explicit or implicit)
Assess whether the software is suitable to meet the customers' needs	Give management the information it needs to assess potential risks associated with the product
Calibrate performance	Measure processing speed, response time, resource consumption, throughput and efficiency
Measure reliability	Quantify the reliability of the software for the customer (reliability demonstration), or for internal improvements (reliability growth) prior to delivery to the customer
Ensure changes/modifications have not introduced new faults	Referred to as regression testing
Establish due diligence for protection against product liability litigation	May provide some level of protection against (justifiably or unjustifiably) dissatisfied customers

The effectiveness of software testing methods, whether they are for detection or demonstration, is directly influenced by the characteristics of the software. Software whose characteristics directly relate back to clear, specific requirements is said to be testable, and the ability of the software to be effectively tested is referred to as its "testability". Testability can relate to either:

- The degree to which a stated requirement allows test criteria and test performance to be defined in order to determine whether the criteria have been met, or
- The degree to which a system or component is designed so that test criteria and performance of tests can be efficiently defined to determine whether the criteria have been met

Table 3.5.1-2 provides an overview of those characteristics that, if applied in practice, can lead to highly testable software.

Table 3.5.1-2: Characteristics of Testable Software

Characteristic	Comment
Operability	**"The better it works, the more efficiently it can be tested"** – implies that the software has few defects, thereby reducing the analysis and reporting burden during testing. This also implies that defects that do exist do not interfere with the execution of tests. The evolution of a product in functional stages allows simultaneous development and testing.
Observability	**"What you see is what you test"** – A distinct output is generated for each unique input. Past and present software states/variables are visible or can be queried during execution. All factors affecting the output are visible. Incorrect outputs are easily identified. Internal errors are automatically detected and reported. Source code is accessible.
Controllability	**"The better the software can be controlled, the more the testing can be automated/optimized"** – All possible outputs can be generated, and all code is executable, through some combination of inputs. Software and hardware states/variables can be directly controlled by the test engineer. Input/output formats are consistent and structured. Tests can be conveniently specified, automated and reproduced.
Decomposability	**"By controlling the test scope, problems can be more quickly isolated and smarter re-testing can be performed"** – The system software is built from independent modules that can be tested independently.
Simplicity	**"The less there is to test, the more quickly it can be tested"** – Functional simplicity (minimum feature set to meet requirements); Structural simplicity (modular architecture to minimize fault propagation); Code simplicity (adopted coding standard eases inspection/maintenance).
Stability	**"The fewer the changes, the fewer the test disruptions"** – Software changes are infrequent, controlled, and do not invalidate existing tests. Software recovers well from failures.
Understandability	**"The more information we have, the smarter we will test"** – The design is well understood. Dependencies between internal, external and shared components are well understood. Design changes are effectively communicated. Technical documentation is instantly accessible, well organized, accurate, specific and detailed.

Adapted from Reference 3.

Figure 3.5.1-1 provides a very generic overview of the overall definition and implementation of a testing process for software. There are a number of different types of specific, dedicated software testing that should be considered in the context of achieving optimized software reliability. These include:

- Control-Flow Testing
- Loop Testing
- Data Flow Testing
- Transaction-Flow Testing
- Domain Testing
- Finite-State Testing
- Orthogonal Array Testing
- Statistical Usage Testing
- Operational Profile Testing
- Markov Testing
- Optimal Release Time

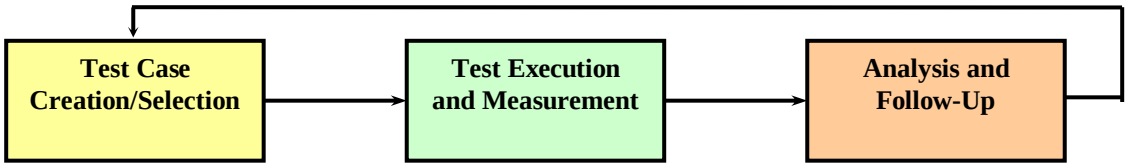


Figure 3.5.1-1: Generic Testing Process for Software

1. Internal Program Information
2. External Specification/Requirement
3. Creation/Selection/Generation
4. Execution (Normal vs. Abnormal)
5. Data Capturing/Other Analysis
6. Result-Checking and Analysis
7. Defect Removal
8. Test Process Improvement

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Dunn, R.H.; Ullman, R.S., "TQM for Computer Software", McGraw-Hill, 1994, ISBN 007018314-7
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783
4. DACS Software Testing Resource Page, http://dacs.dtic.mil/databases/url/key.hts?keycode=2399
5. Software Testing Hotlist, http://www.io.com/~wazmo/qa/

Topic 3.5.2: Software Test Coverage Metrics

Test coverage relates to the proportion and type of testing that is performed on software at any level of complexity and the software development life cycle phase during which testing occurs. Table 3.5.2-1 summarizes the types of tests comprising test coverage that may be performed on the software product. An effective test coverage strategy will have a direct positive impact on software reliability, as more testing will increase the probability that defects will be found and corrected. A strategic goal, however, should always be to design/develop software that it is inherently reliable, i.e., high reliability is most efficiently achieved through good design practice and continuous process improvement, rather than through extensive and expensive testing regimes.

Table 3.5.2-1: Test Types Comprising Test Coverage

Test Type	Characteristics
Clear box	Focuses on "how" the software works (equivalent to structural testing). Sometimes called "glass-box testing". Uses the control structure of the procedural design to derive test cases that (1) guarantee all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true/false sides, (3) exercise all loops at and within their operational boundaries, and (4) exercise internal data structures to ensure their validity. Should be performed early in the testing process
Black Box	Tests software with respect to its external requirements and specifications (synonymous with functional and behavioral testing). Independent of program size or level. Uses representative data as inputs, and outputs are compared to the requirements/specs. Tests focus on what the software is supposed to do (i.e., the information domain), not "how" the software works. Attempts to find errors due to (1) incorrect/missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavioral or performance errors, and (5) initialization and termination errors.
Unit	The unit testing process (a clear-box application) focuses on the internal logic of the software, making sure that all statements have been tested.
Integration and System	The integration and system testing process (a black-box application, with some elements of clear-box testing) focuses on the external function of the software, testing to uncover errors and to ensure that the defined input will produce actual results that agree with required results.
Acceptance	The acceptance testing technique (a black-box application) uses independent test teams (i.e., not the software development team) to examine the completed system to determine if original functionality requirements have been met.

Advocates of test coverage reliability metrics have defined software reliability as a function of the amount of software product that has been successfully verified or tested. Their rationale is that, since the data are (or should be) collected and tracked during testing, the test coverage metrics should be readily available with no additional verification effort required. Note, however, that:

- Test coverage reliability metrics are ***not*** commonly used or understood by traditional reliability practitioners and program managers
- Test coverage reliability metrics cann***ot*** be converted to failure rates or used to predict/estimate mean time to failure (MTTF) or mean time between failure (MTBF)

Test Success Reliability Metric

Reliability is defined as the ratio of the number of test cases successfully completed during Acceptance testing to the total number of test cases executed during Acceptance testing, given as:

$$\mathbf{R} = \mathbf{s}/\mathbf{r}$$

where,

R = test coverage reliability

s = number of test cases successfully completed
r = total number of test cases executed

If 95 out of 100 test cases were completed successfully, then the test success reliability would be 0.95, or 95%.

Note that this metric is a function of only those test cases that have been *executed*. If test cases have not been thoroughly defined there may be a number of relevant cases unintentionally ignored that may have failed had they been executed. The validity of this metric is based on the skill with which test cases are defined and executed.

IEEE Test Coverage Reliability Metric

This method assumes that reliability is dependent on both the functions that are tested (black-box) and the product that is tested (clear-box). In order for test coverage to be complete, it is assumed that both types of testing have been performed. The test coverage reliability is computed as:

$$\mathbf{R = P_F * P_P}$$

where,

R = test coverage reliability
P_F = ratio of the total number of capabilities tested to the total number of capabilities inherent in the software
P_P = ratio of the total number of paths/inputs tested to the total number of paths/inputs inherent in the software

If the software inherently contains 10 capabilities, of which 9 are tested, then $P_F = 9/10$, or 0.90. Similarly, if the software inherently contains 50 paths/inputs, of which 48 are tested, then $P_P = 48/50$, or 0.96. The combined test coverage reliability, R, is the product of the two, or 0.90*0.96 = 0.864.

Leone's Test Coverage Reliability Metric

This metric is similar to the IEEE metric described above, except that it assumes that it is possible to perform either white or black box testing and still achieve some level of test coverage reliability. Using this technique, two clear-box proportional variables and two black-box proportional variables are defined. The test coverage reliability is the weighted sum of these four proportions, as given below:

$$\mathbf{R = \frac{(a*w_1)+(b*w_2)+(c*w_3)+(d*w_4)}{w_1+w_2+w_3+w_4}}$$

where,

R = test coverage reliability
a = ratio of the total number of independent paths tested to the total number of paths inherent in the software
w_1 = weighted importance of the factor "a"
b = ratio of the total number of inputs tested to the total number of inputs inherent in the software
w_2 = weighted importance of the factor "b"
c = ratio of the total number of functions verified to the total number of functions inherent in the software
w_3 = weighted importance of the factor "c"
d = ratio of the total number of failure modes addressed to the total number of failure modes inherent in the software
w_4 = weighted importance of the factor "d"

The values for w_1 through w_4 represent weights. If factors "a" through "d" are of equal importance, the weights should all be set to 1. If, however, there is information (data or judgment) that supports the premise that some parameters are more important than others, then those parameters should be weighted heavier (and normally the sum of w_1 through w_4 should equal 1).

This metric assumes that (1) independent paths are identified using McCabe's complexity methodology, (2) inputs are identified using the information domain structure defined for Function Points analysis, and (3) software failure modes are identified using Fault Tree Analysis (FTA) or Failure Modes, Effects and Criticality Analysis (FMECA).

As an example, assume that the following ratios have been determined for each of the four proportional variables:

a = 0.96
b = 0.98
c = 0.99
d = 0.95

For the purposes of test coverage reliability, it has been analytically determined that the total number of failure modes addressed (parameter "d") is the most important. The total number of inputs tested (parameter "b") and the total number of functions verified (parameter "c") are equally important. Of "least" importance is the total number of independent paths tested (parameter "a"). One weighting scheme that could be assigned is:

w_1 = 0.10
w_2 = 0.15
w_3 = 0.15
w_4 = 0.60

The resulting test coverage reliability is calculated to be:

$$\mathbf{R} = \frac{(0.96*0.10)+(0.98*0.15)+(0.99*0.15)+(0.95*0.60)}{0.10+0.15+0.15+0.60} = 0.9615$$

A second weighting scheme of $w_1 = 0.05$, $w_2 = 0.25$, $w_3 = 0.25$, and $w_4 = 0.45$, using the same values for the four proportional variables, provides different results:

$$\mathbf{R} = \frac{(0.96*0.05)+(0.98*0.25)+(0.99*0.25)+(0.95*0.45)}{0.05+0.25+0.25+0.45} = 0.968$$

Compare the two weighted results with the test coverage reliability when all factors are weighted equally:

$$\mathbf{R} = \frac{(0.96*1)+(0.98*1)+(0.99*1)+(0.95*1)}{1+1+1+1} = 0.97$$

For More Information:

1. MIL-HDBK-338 "Electronic Reliability Design Handbook", Section 9.5.2.4
2. Neufelder, A.M., "Ensuring Software Reliability", Marcel Dekker, Inc., 1993, ISBN 0824787625
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.3: Control-Flow Testing

Control-flow testing can be defined and implemented as either a structural (clear-box) or behavioral (black-box) testing strategy, depending on the level of information available regarding the software, or the resources available for developing and performing the tests (i.e., time and money). Table 3.5.3-1 directs the reader to the appropriate references for a more detailed discussion for each strategy. Table 3.5.3-2 summarizes some of the basic characteristics of clear-box versus black-box control-flow testing.

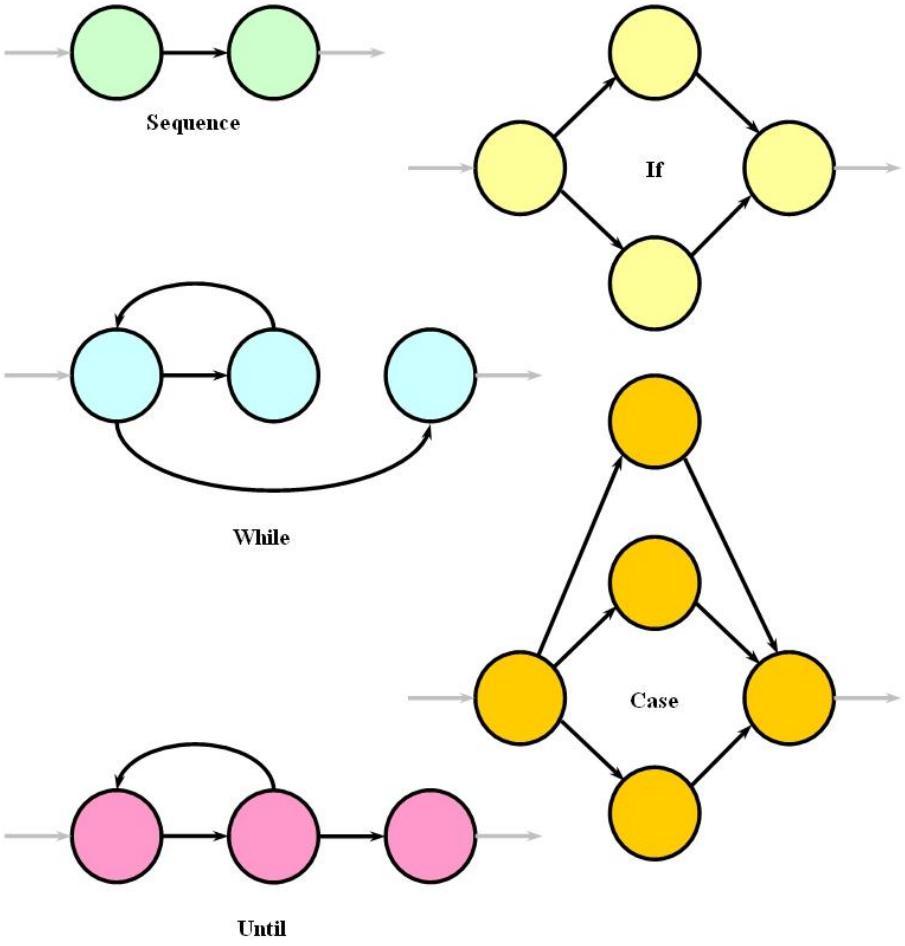
Table 3.5.3-1: Discussions of Control-Flow Testing in the Literature

Reference	Test Class	Synonyms	Comments
Pressman (Ref. 3, Sect. 17.4)	Clear-box	Basis Path	Enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. These test cases are guaranteed to execute every statement in the program at least once during testing.
Lyu (Ref. 2, Sect. 13.2.2)	Clear-box	Statement Coverage Decision Coverage	Testing directs the tester to construct test cases such that each statement or basic block of code is executed at least once Testing directs the tester to construct test cases such that each decision in the program is covered at least once
Beizer (Ref. 1, Chapter 3)	Black-box	Behavioral Control Flow	Considered to be the fundamental model of black-box testing, basic to all other black-box testing techniques

Table 3.5.3-2: Clear- and Black-Box Characteristics of Control-Flow Test Graphs

Characteristic	Clear-box	Black-Box
Source for testing	Design/code	Specification
Control flow graphs:		
- Nodes	Assignments and calls	"Do" (enter, calculated, etc.)
- Branches	"Go to/if/when/while/…"	"Go to/if/when/while/…"
- Loops	"Repeat" (for all, until, etc.)	"Repeat" (for all, until, etc.)
- Entry	Explicit	Usually implicit
- Exit	Explicit	Implicit and explicit

Figure 3.5.3-1 illustrates notation typically used to represent logical control-flow in a flow graph (sometimes referred to as a program graph). The application of this notation in the use of a control-flow graph is shown in Figure 3.5.3-2.



Note: Each circle represents one or more non-branching program decision language (PDL) or source code statements. PDL is also referred to as pseudocode.

Figure 3.5.3-1: Typical Notation for Logic Control Flow

The basic definitions corresponding to the elements of a control graph are defined in Table 3.5.3-3.

Table 3.5.3-3: Definitions of Control Graph Elements

Element	Definition
Node	A node, represented by a circle, is a representation of one or more procedural statements. A sequence of process boxes and a decision diamond may map into a single node.
Predicate Node	A predicate node is any node that contains a condition that selects one of two or more alternate paths that a process can take, e.g., "IF x OR y". A predicate node is characterized by two or more links (edges) exiting from the node.
Link (or Edge)	A link, represented by an arrow, represents the flow of control between nodes. A link represents a graphical representation of the relation between the connected nodes. A link must terminate at a node.
Region	A region represents an area bounded by edges and nodes. Regions are used in the determination of the quantitative cyclomatic complexity measure of a program's logical complexity.

Figure 3.5.3-2(a) represents a flowchart that is used to represent a hypothetical control structure. Figure 3.5.3-2(b) maps that structure into its corresponding control graph.

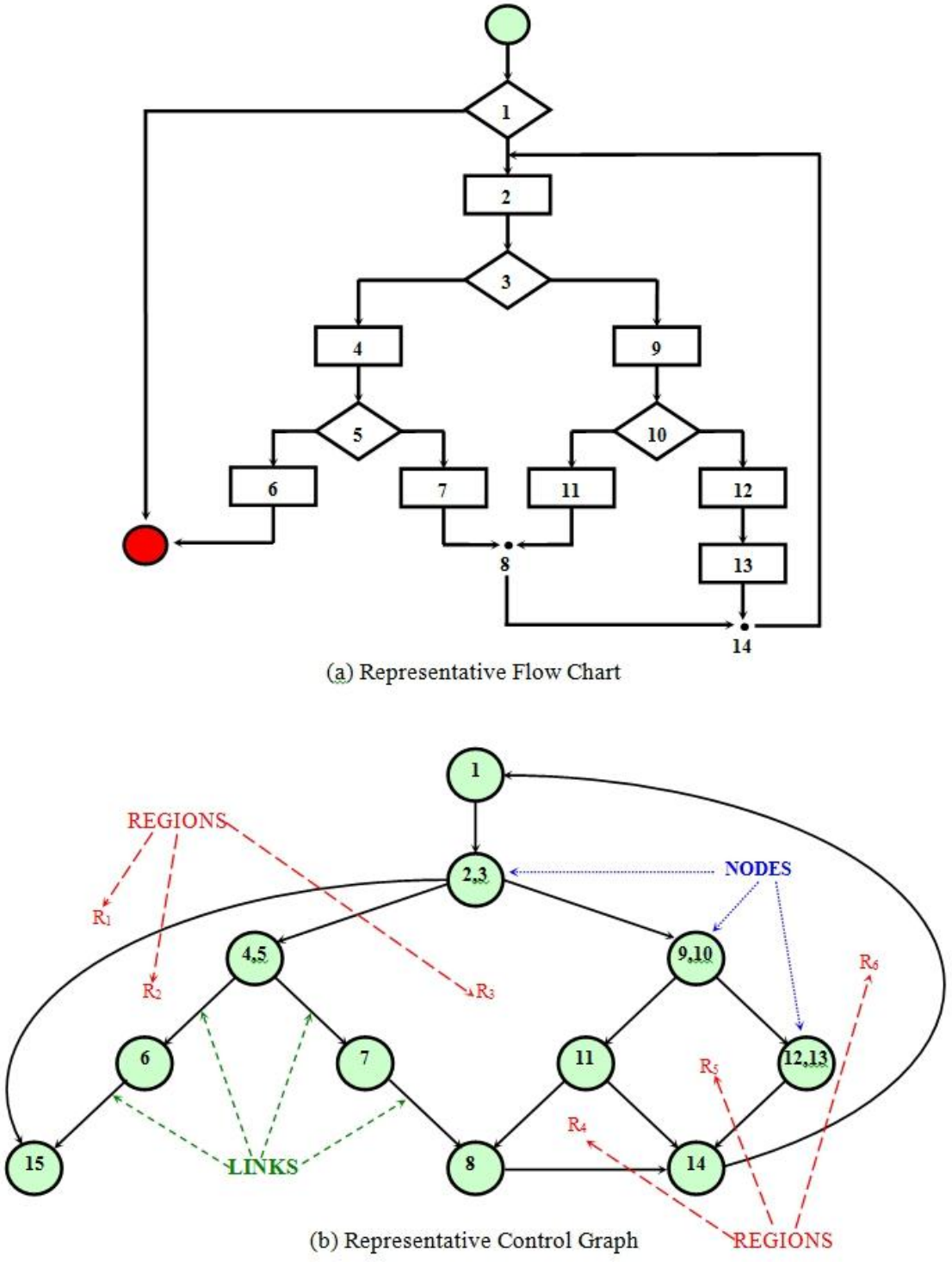


Figure 3.5.3-2: Example of Mapping a Flow Chart Into a Control Graph

Table 3.5.3-4 describes the basic steps for performing Control-Flow testing, covering both the clear-box and black box scenarios. Note, however, that many of the characteristics of the basic steps are common between the two processes.

Table 3.5.3-4: The Control-Flow Testing Process

Step	Discussion
1. Preparation	For the clear-box test, based on the software design/code, develop a flowchart that defines all of the elements contained within the program. For the black-box test, examine and analyze the requirements for completeness and self-consistency. Confirm that the specification correctly reflects the requirements. Rewrite the specification as a sequenced list of short sentences, paying close attention to predicates. Uniquely number each sentence, as they will become node names.
2. Build the model	For the clear-box test, using the notation presented in Figure 3.5.3-1, construct the control-flow graph (see Figure 3.5.3-2 for an example). For the black-box test, construct the control-flow graph. Beizer (Reference 1) suggests that list notation is more convenient than graphs, but the use of small graphs can aid in the design of the model. Compound predicates should be avoided in the model, and replaced by equivalent graphs so that essential complexity is not hidden. Use of a truth table is recommended instead of a graph when trying to model compound predicates with more than three component predicates. The model should be segmented into pieces that start and end with a single node. Also, note which predicates are correlated with each other in all other segments.
3. Verify the model	Verify the model through self-testing to ensure that the model itself does not contain any defects.
4. Define/select test paths	Define and select enough paths through the model to ensure that every link is tested at least once. Start by picking the obvious paths that are independent, i.e., that move along at least one edge that has not previously been covered (clear-box tests) or paths that relate directly to the requirements, augmenting them with however many paths may be needed to guarantee 100% link coverage (black-box tests).Cyclomatic complexity is precisely the minimum number of paths that can, in linear combination, generate all possible paths through the module. Therefore, cyclomatic complexity measures can be used to determine the minimum number of paths that should be tested. Complexity is computed one of three ways: 1. Cyclomatic complexity = The number of regions in the control-flow graph 2. Cyclomatic complexity = (Number of flow graph edges) – (Number of flow graph nodes) + 2 3. Cyclomatic complexity = (Number of predicate nodes in the flow graph) + 1 From the control-flow graph in Figure 3.5.3-2(b), there are 6 regions (method 1); there are 15 edges and 11 nodes, so that (15-11+2) = 6 (method 2); and there are 5 predicate nodes, so that (5+1) = 6 (method 3). A graph matrix is a tabular representation of the control-flow graph. This matrix, whose number of rows and columns equals the number of nodes in the control-flow graph, and whose matrix entries correspond to a link, or edge, connecting the nodes, can be used to determine an effective set of paths to be tested. By adding a link weight to each matrix entry, the matrix can become a tool for evaluating program control structure. Link weights can provide additional information about control-flow, e.g., the probability that a link will be executed; the processing time expended during link traversal; the memory required during link traversal; or the resources required during link traversal. A matrix containing link weights is referred to as a “connection matrix”. The control-flow graph of Figure 3.5.3-2(b) is used to illustrate this concept in Table 3.5.3-5. A node connection is entered into the matrix as a “1” (or TRUE), and a blank entry in the matrix implies a “0” (or FALSE). Note that this type of connection matrix can be used to determine the number of predicate nodes in the control graph and, subsequently, the cyclomatic complexity of the program. From the example, there should be six linearly independent paths through the program control-flow. Test cases should be prepared that will force execution of each of the six paths.

Table 3.5.3-4: The Control-Flow Testing Process (continued)

Step	Discussion
5. Sensitize the selected test paths	Sensitization is defined as the use of input values that will cause a selected path in the model to be traversed, assuming that there are no defects in the model implementation. The sensitizing procedure is dependent on the nature of the predicates along the path being tested. If the predicates are primarily logical, sensitization is typically performed concurrently with path selection. If the predicates along the path are predominantly numeric or algebraic, then a different process will be used. Sensitize the appropriate paths to be tested by interpreting the input values of the predicates along the path. These interpreted predicates result in a set of conditions and mathematical inequalities that will provide a solution set such that specific solution in that set will cause the selected path to be traversed.
6. Predict/ record expected outcome for each test	The expected outcome for each selected path can be predicted using a variety of alternatives (see Reference 1): Existing Tests: Most testers/programmers work on modifications to an existing base of software, meaning that many tests will not need to change between releases. If tests are kept under strict configuration control, then they can serve as the foundation for most of the new tests. Old Program: A major program rewrite may not require equivalent changes to its associated test suite, so that the old program may serve as an oracle, e.g., an existing program written for one platform may need to be ported to other mainframes. Although the re-hosting may require extensive rewrite, the old program can be run with the new tests to determine predicted outcomes. Previous Version: Even if the code being tested represents a complete rewrite, a previous version may have the correct outcome for most test paths. Use outcomes from the previous version as a starting point for finding the results of the present version on corresponding paths. Prototypes/Model Programs: Prototypes that are otherwise too big, too slow, or will not run in the targeted environment may still possess enough functionality to provide a correct expected outcome. If a prototype doesn't exist, a model program can be built. The model program would only need to address the program logic and algebra and would not need to be concerned with access to data structures, operating system interfaces, inputs and outputs. Forced Easy Cases: It may be possible to select unrealistic input values that force a traverse along a selected path, but that are trivial to calculate. Note that realistic input values are important for demonstrating program capabilities, but not necessarily very good at revealing defects. If allowable input values for testing are expanded to include unrealistic values, the process of output prediction, predicate interpretation and sensitization can be less painful. Actual Program: It's typically easier to verify the correctness of an outcome than it is to manually simulate the computer calculation to determine the outcome, especially if there are verifiable intermediate values that are made available. The assumption is that the analysis required to verify the outcome will actually be performed, rather than the outcome accepted as is, without verification.
7. Define the validation criteria for each test	Before testing commences, define and document what the validation criteria will be for each test performed, i.e., what are the outcome results that will be acceptable as an indicator that the test has been successfully passed (pass/fail criteria). Define how normal and failure case scenarios will be handled.
8. Perform the tests	Automate, automate, automate
9. Confirm each test result	Compare test results with those predicted, and with the defined validation criteria, to determine whether a test has passed or failed. For failed tests, proceed with root-cause analysis and corrective action identification, implementation and verification. For tests that have passed, go to Step 10….
10. Verify the path	Path verification is needed to avoid the pitfalls of potential "coincidental correctness". It is not necessary to verify every computation, but as much of the path should be verified as is convenient given the available resources and constraints. Another guard against coincidental correctness is to test several cases along each defined path. This happens as a natural result of other test techniques, particularly domain testing.

Table 3.5.3-5: Example Connection Matrix

Node	Connection to Node											Connections
	1	(2,3)	(4,5)	6	7	8	(9,10)	11	(12,13)	14	15	
1		1										1 – 1 = 0
(2,3)			1				1				1	3 – 1 = **2**
(4,5)				1	1							2 – 1 = **1**
6											1	1 – 1 = 0
7						1						1 – 1 = 0
8										1		1 – 1 = 0
(9,10)								1	1			2 – 1 = **1**
11						1				1		2 – 1 = **1**
(12,13)										1		1 – 1 = 0
14	1											1 – 1 = 0
15												0 – 0 = 0
							Cyclomatic complexity =					**5 + 1 = 6**

For More Information:

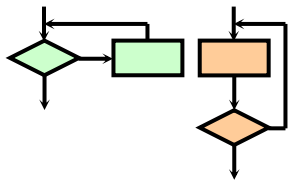
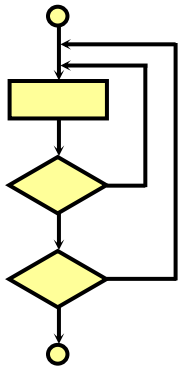
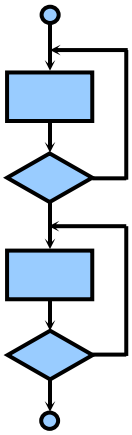
1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783
4. The Machine-SUIF Control Flow Graph Library, http://www.eecs.harvard.edu/hube/software/nci/cfg.html
5. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric (NIST Special Publication 500-235), http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm

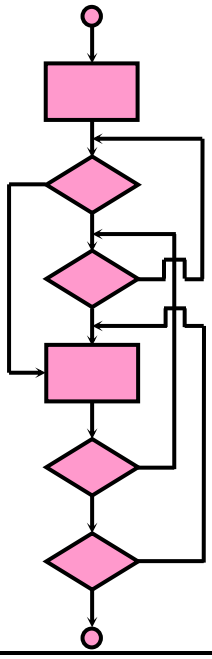
Topic 3.5.4: Loop Testing

Program loops represent the foundation for the vast majority of all algorithms contained within software. Loop testing provides focus on the validity of loop structures. Virtually every repetitive process should be exposed to loop testing. The only possible exceptions to this rule-of-thumb are finite-state testing (where there are too many loops to test adequately) and domain testing (where the mere presence of loops makes the technique impractical). The importance of loop testing is based on the fact that programs generally tend to contain a relatively high number of defects associated with starting and stopping loops.

Table 3.5.4-1 provides an overview of the four basic classes of loops in the context of clear-box testing.

Table 3.5.4-1: Basic Classes of Loops (from Reference 3)

Class	Comments	Graphical Representation
Simple	The following sets of tests are applicable for simple loops. The variable "n" is defined as the maximum number of allowable passes through the loop. - Skip the loop entirely - One pass only through the loop - Two passes through the loop - "m" passes through the loop (m<n) - "n-1", "n", and "n+1" passes through the loop	
Nested	Extending the test philosophy for simple loops to nested loops would result in an impractical number of tests. Reference 1 suggests, instead: - Start at the innermost loop, setting all other loops to minimum values - Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum loop counter values. Add other tests for out-of-range or excluded values. - Work outward, conducting tests for the next loop, keeping all other outer loops at their minimum values, and lower-level loops at their typical values - Continue these steps until all loops have been tested	
Concatenated	These loops can be tested using the same approach defined for simple loops if each of the loops is independent of the other. If, however, two loops are concatenated and the loop counter for one is used as the initial value for the other they are not independent and the test approach for nested loops should be applied.	

Class	Comments	Graphical Representation
Unstructured	Unstructured loops come about when a program jumps out of, or into, the middle of a loop. The loop entry node for the first iteration is not the same as the loop entry node for subsequent iterations. There are no good tests for unstructured loops. Unstructured loops must be tested more carefully and thoroughly than normal due to their susceptibility to being improperly implemented. Whenever possible, unstructured loops should be redesigned using basic structured programming constructs.	

Reference 2 discusses loop testing in the context of black-box tests, including the concepts of deterministic and nondeterministic loops.

A deterministic loop is one whose iteration count is known before the execution of the loop begins. In addition, there is no processing being performed within a deterministic loop that will cause that number to change (i.e., the iteration count remains fixed). Looping processes should be constructed using deterministic loops when (1) copying a file with a known number of records, (2) processing "n" number of payroll checks, (3) adding a column of numbers, (4) filling an array with numbers, and (5) transmitting a file of known length.

A nondeterministic loop is one whose iteration count is unknown before the execution of the loop begins, or a loop whose iteration count is defined or changed by processing that occurs after the loop has been entered (i.e., the iteration count may be variable). Nondeterministic loops tend to have more defects than deterministic loops and, therefore, need to be tested more thoroughly.

Table 3.5.4-2 defines the generic test cases, and their critical values, that should be used for performing loop testing, regardless of whether the process is structural or behavioral. Critical test values, in addition to the normal or typical case, are defined to be the combination of values of the starting value of the loop control variable, the ending value of the loop control variable and the size of the increment (number of steps) of the loop control variable for each pass through the loop. For example, given a loop with the statement "**FOR I = 0 to 10 STEP 3**", the critical test value is 4, since that is the number of times the loop will be executed based on incremental steps of 3. For testing nested loops, these values would be tested in combination using the guidelines discussed in Table 3.5.4-1.

Table 3.5.4-2: 12 Generic Test Cases/Critical Values for Loop Testing

Test Case	Critical Values	Upper Bound	Lower Bound	Other
Bypass	Any value that causes the loop to be exited immediately		X	
Once	Values that cause the loop to be executed exactly once		X	
Twice	Values that will cause the loop to be executed exactly twice		X	
Typical	A typical number of iterations			X
Maximum	The maximum number of allowed loop iterations	X		
Max. + 1	One more than the maximum number of allowed iterations	X		
Max. –1	One less than the maximum number of allowed iterations	X		
Minimum	The minimum number of required loop iterations		X	
Min. + 1	One more than the minimum number of required iterations		X	
Min. –1	One less than the minimum number of required iterations		X	
Null	A value "zero" may or may not be redundant with the "Bypass" test case			X
Negative	A value less than zero that may impact output integrity for the next dataset			X

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.5: Data-Flow Testing

Data-flow testing, as described by Pressman (Reference 3) selects test paths of a program according to the location and definitions and uses of variables within the program. It is a more powerful technique than control-flow testing. Beizer (Reference 1) points out that, since control-flow testing can be considered a subset of data-flow testing, all of the defects precipitated in the former will be precipitated in the latter. There is a basic assumption, however, that programmers will have been able to eliminate simple control-flow defects from their programs. Since data-flow test models should avoid unessential control flows, the basic type of defects that can be expected will tend towards data defects such as those associated with initial and default values; duplication and aliases; overloading; wrong item; wrong type; bad pointers; and data-flow anomalies (such as closing a file before opening it).

Data-flow testing is based on first defining a data-flow model, and then using that model to develop effective test designs. Tests are created by selecting "slices" from the output nodes to all of the corresponding input nodes of the slice.

Summarizing from Reference 3, the test design and execution process is the same as for the control-flow testing approach discussed in Section 3.5.3. There are some minor differences that result based on the fact that the process is using data-flow graphs, as opposed to control-flow graphs. The different types of nodes and links that can make up a data-flow graph are summarized in Table 3.5.5-1 (adapted from Reference 1). Table 3.5.5-2 illustrates the steps associated with data-flow testing.

Table 3.5.5-1: Node and Link Definitions for Data-Flow Graphs

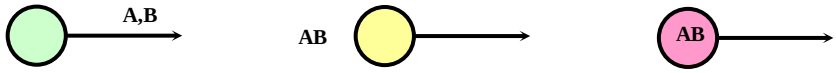
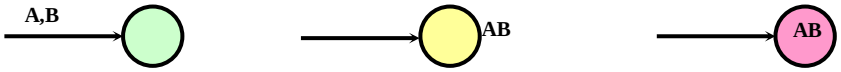
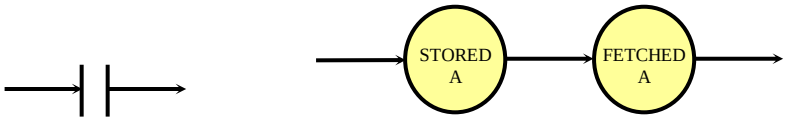
Element	Comments
Input Node	An entry node of a data-flow graph model through which data are input. The name of the object input at that node is usually written in the node or just preceding the node, but it may also be written on the node output link, as illustrated below. 
Output Node	An exit node of a data-flow graph model through which data is output. The name of the object whose value is output can be written in, preceding, or next to the output node. 
Storage Node	Represents a pair of nodes for the same data object. The STORE node defines the value of the stored variable. The FETCH node defines the value of the variable in memory. The symbol on the left represents the way that a storage node is typically shown in data-flow graphs, but the symbol on the left is clearer. 

Table 3.5.5-1: Node and Link Definitions for Data-Flow Graphs (continued)

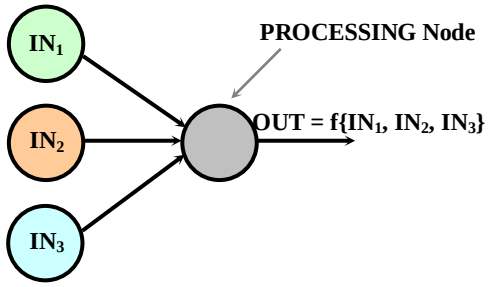
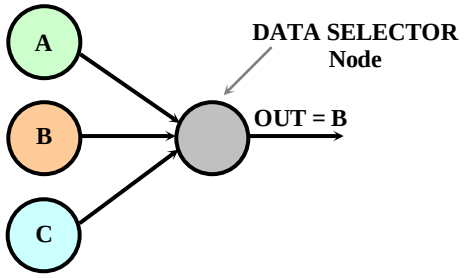
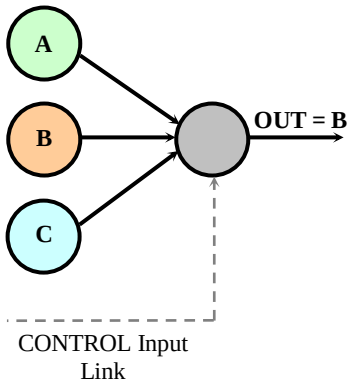
Element	Comments
Processing Node	A node with one or more input links, and at least one output link. The input links represent data objects, while the output link represents a calculated function of those data objects. In the example, OUT is a function of IN_1, IN_2 and IN_3. 
Data Selector Predicate	A predicate whose value is used to select one of multiple data objects. There is always a data selector predicate for a data selector node.
Data Selector Node	A node whose input links are controlled by a data selector predicate. The value selects a data object that is associated with an input link. A data selector node calculates the special function of selecting the input link value that will be used as the output link. The input links of a data selector node can be annotated with the predicate condition that selects that input link. 
Control Inlink	A data selector node may have an input link whose use is exclusively defined by the data selector predicate. A common convention identifies control links through the use of dashed lines. The selected value at the output link (B) could be based on the value at a control input link that specifies which input link value is to be used, such as in a pointer to an array. 

Table 3.5.5-2: The Data-Flow Testing Process

Step	Discussion
1. Preparation	Examine and analyze the requirements for completeness and self-consistency. Confirm that the specification correctly reflects the requirements. Identify all input variables, particularly constants. Rewrite the specification as one sentence per function that is to be calculated. Uniquely name each input variable and assign it an input node.
2. Build the model	Begin by listing the defined functions, starting with those that depend only on the input variables, then proceeding to those functions whose input variables depend only on the outputs from the previous functions. Continue listing each of the defined functions in this manner until all are accounted for. The end result should be a list of functions such that the first set of functions will depend only on input variables, and subsequent functions on the list will depend increasingly on intermediate calculations (i.e., those that depend on both input variables and output results). All intermediate functions should be assessed to see if the sequencing of functions in the list is essential, or just convenient. If they are essential, that node (and the nodes on which it depends) should be labeled as such. If the sequence of the functions is not essential, the model can possibly be simplified by removal of the appropriate intermediate nodes and expressing the function explicitly in terms of its input variables. Other nodes and links can possibly be removed to simplify the model, as long as the calculated function does not become overly complicated. The model could, conversely, be simplified by actually adding an intermediate node(s) for a difficult calculation (understanding that it will be necessary to verify that the new intermediate calculation is correct). The result of this process should be a set of nodes, each with a name that expresses the data-flow in a way that is conceptually easy to understand. There is now a computation or function associated with each of the defined nodes. The variables within each function name the nodes to which these nodes are connected (i.e., the links).
3. Verify the model	Verify the model through self-testing to ensure that the model itself does not contain any defects.
4. Define/select test paths	The process for defining/selecting data-flow test paths is similar to that of control-flow testing, except for some minor differences in the step details, requiring an understanding of some additional definitions (per Reference 1): - **Subgraph:** A part of a graph that conforms to the standard rules of flow graphs (e.g., the presence of entry/exit nodes; no dangling links; no unconnected nodes, etc.) - **Slice:** A subgraph that is selected based on conformance to a predefined criterion such that, for that criterion, the subgraph reflects all of the properties of the entire graph for the selected nodes and links. There are many different kinds of slices, based on many different sets of criteria. For control-flow testing, the slice would encompass the model of all the relevant code on the selected path. For data-flow testing, the greatest interest is in data-flow slices. - **Data-Flow Slice:** These are taken with respect to data objects. In general, a data-flow slice with respect to a given node (object) represents a subgraph of the overall data-flow graph, which consists of all of the data flows that can directly or indirectly reach the specified node, plus all data flows that can be reached from that node. If a slice is with respect to an output node, as is typical, then it includes (if defined properly) all of the nodes than can influence the value of that output. In a practical sense, the process is to trace back from the node of interest and "label" any input links into that node, then input links into the nodes that those links came from, and so on, and similarly for link outputs from the node of interest.

Table 3.5.5-2: The Data-Flow Testing Process (continued)

Step	Discussion
5. Sensitize the selected test paths	Sensitization is defined as the use of input values that will cause a selected path in the model to be traversed, assuming that there are no defects in the model implementation. The sensitizing procedure is dependent on the nature of the predicates along the path being tested. If the predicates are primarily logical, sensitization is typically performed concurrently with path selection. If the predicates along the path are predominantly numeric or algebraic, then a different process will be used. Sensitize the appropriate paths to be tested by interpreting the input values of the predicates along the path. These interpreted predicates result in a set of conditions and mathematical inequalities that will provide a solution set such that specific solution in that set will cause the selected path to be traversed. **For data-flow testing, it may be easier to start at the output and work "upward" to the inputs. If the data-flow slice does not contain selectors or control-flow nodes, any acceptable input values will work, i.e., there is no significant sensitization.**
6. Predict/record expected outcome for each test	Although the principles of predicting the expected outcome for each data-flow test are analogous to those described for control-flow testing (see Section 3.5.3), the fact of the matter is that, since there shouldn't be much control-flow in a data-flow model, a spreadsheet is a reasonable choice for building an oracle. Each cell of the spreadsheet would represent an obvious node, and direct data-flow relationships would be addressed through the formulas in the cell.
7. Define the validation criteria for each test	Before testing commences, define and document what the validation criteria will be for each test performed, i.e., what are the outcome results that will be acceptable as an indicator that the test has been successfully passed (pass/fail criteria). Define how normal and failure case scenarios will be handled.
8. Perform the tests	Automate, automate, automate
9. Confirm each test result	Compare test results with those predicted, and with the defined validation criteria, to determine whether a test has passed or failed. For failed tests, proceed with root-cause analysis and corrective action identification, implementation and verification. For tests that have passed, go to Step 10….
10. Verify the path	Path verification, in this case "node verification", is needed to avoid the pitfalls of potential "coincidental correctness". It is not necessary to verify every computation, but as many nodes as possible should be verified as is convenient given the available resources and constraints. Another guard against coincidental correctness is to test several cases for each defined node.

Beizer (Reference 1) identifies several different scenarios for considering slice selection, depending on how the data-flow graph is defined. An overview of these 5 scenarios is provided in Table 3.5.5-3 on the next page.

Table 3.5.5-3: Scenarios for Determining Data-Flow Slice Selection

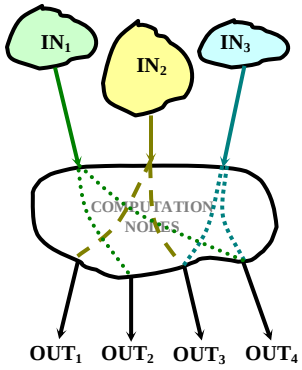
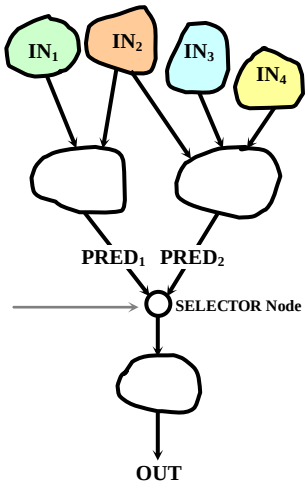
Scenario	Comments	Sample Data Flow
Pure Data Flows	For every output node, trace backward from that node to all nodes that connect to it. Trace back from those nodes to all nodes connecting to them, etc., until you have reached input nodes for that model. The result is a data-flow slice. From the example, there are 3 input and 4 output variables. OUT_1 depends only on IN_2 variables. OUT_2 depends only on IN_1 variables. OUT_3 depends on IN_2 and IN_3 variables, while OUT_4 depends on IN_1 and IN_3 variables. There are four sets of tests corresponding to the four output variables and, with no selector or control-flow nodes, exactly one test case per output variable. Note that all of the outputs may have some, but probably not all, computation nodes in common (i.e., none are fully independent data-flow paths).	
Data Flows and Selectors Only	Start with a slice for every output variable, as above. When a selector node is reached, however, every potentially selected test case must be included in the slice. Each of these "superslices" will result in a set of test cases, and each superslice must be considered one at a time. Assume that there's only a single selector in the slice. For each value of the selector predicate, select a value and then exclude all data flows from the superslice that do not contribute to the determination of that value. The slice based on OUT includes 3 computation nodes areas, the dataset inputs IN_1, IN_2, IN_3 and IN_4, and the SELECTOR node. Picking the $PRED_1$ value, which depends only on IN_1 and IN_2, excludes IN_3, IN_4 and $PRED_2$ from the slice. Picking the $PRED_2$ value excludes IN_1 and $PRED_1$, since $PRED_2$ (and, hence, OUT) depends only on IN_2, IN_3 and IN_4.	

Table 3.5.5-3: Scenarios for Determining Data-Flow Slice Selection (continued)

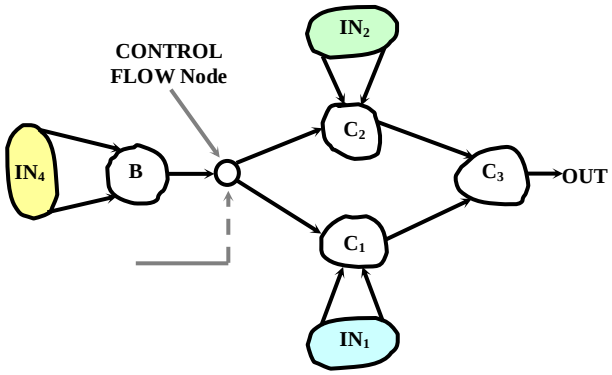
Scenario	Comments	Sample Data Flow
Control-Flow Predicates Without Selectors	Start at the bottom with output variables and create a slice. In a data selector node (previous case) only one input link goes into a slice. In a control flow node, each output link creates a new slice. Start the slice at the output (for each output), encompassing the C_1 and C_2 computation nodes and their (potentially overlapping) IN_1 and IN_2 data sources. At the Control-flow node, however, only the C_1-IN_1 path or C_2-I_2 path will be followed. Each choice defines a test, and both tests require the output from computation nodes set B and its associated data input set IN_4. For Data-Flow Selectors: Slice on the node input links For Control-Flow Nodes: Slice on the node output links	
Mixed Control-Flow Predicates and Data-Flow Selectors	This case is a combination of the two previous cases, so care must be taken in defining slices. Mixed models should be avoided to prevent potential confusion, which won't create a defect, but will waste time and possibly create meaningless tests or tests that can't be executed.	
Loops	Loops are not well suited for data-flow models. The correct approach is to do a complete unfolding for each loop iteration, then put in a data selector for each value. For example, if three test cases were to be chosen they would most likely be looping, looping once, and looping twice. A selector node would need to be added to represent each of the three cases.	

Table 3.5.5-4 presents a progressively more powerful hierarchy of test methods that can be used on data-flow graphs (summarized from Reference 1).

Table 3.5.5-4: Hierarchy of Data-Flow Graph Test Methods

Test Method	Comments
Input/Output Cover	Carefully consider each output node (one for each output). For each output node, use a set of input values that calculates some output value. A weakness is that if a selector predicate is present, only one value of the selector will be used (the others will not be tested). Test method only ensures the program works for one set of input values.
Input/Output + All Predicates	Strengthens input/output coverage by testing all predicates (including control-flow predicates for loops and essential sequencing) using truth values and, analogously, for CASE-statement predicates. Weakness is that there may be intermediate calculations whose values are not used.
Partial Node Cover (All Definitions)	Neither of the previous methods ensured that all nodes/links would be tested. The "All Definitions" strategy ensures that every computational node of the data-flow model has been exercised at least once (implying that intermediate calculations will also be verified). This test method, however, could miss every selector node and loop in the data-flow model.
All Nodes	This step ensures that all nodes are covered, not just computational nodes. As a result, data selector nodes and control-flow nodes will be covered. It does not, however, guarantee that every possibility for selector and control-flow predicates has been checked and verified.
Link Cover (All Uses)	This test method attempts to cover every link in the data-flow graph (corresponding to the "All Uses" strategy). This method verifies every use of a calculated result in subsequent processing, including all intermediate calculations as well as the final outputs. It does not cover every possible path through the program, or even every way that a definition can get to a subsequent use.
All Uses + Loops	An attempt should be made to keep loops out of data-flow models. If that is not possible, the data-flow model should be unfolded and test cases should be augmented by covering the unfolded model's links.
Beizer (Reference 1)	The ultimate test method is somewhere between "All Uses", but not quite as strong as "All-Definition/Use Paths". The intent is to bring in the notion of a simple path without taking all possible DU paths. Unfolding the data-flow model helps to accomplish that goal.

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.6: Transaction-Flow Testing

Information flow in a software system is often characterized by a single data item, defined as a transaction that initiates other data flow along one of many other possible paths. Transaction flow is best characterized by data proceeding along an incoming path that converts information coming in from the outside world into a transaction. This transaction is subsequently evaluated, and where it proceeds next is based on the current transaction value. The center of information activity from which these many action paths can originate is defined as a transaction center. A basic transaction-flow graph is given in Figure 3.5.6-1, and is typically used in system testing of on-line applications and batch processing software. The transaction-flow graph contains both control-flow and data-flow attributes. The different types of nodes and links that can make up a transaction-flow graph are summarized in Table 3.5.6-1 (adapted from Reference 1).

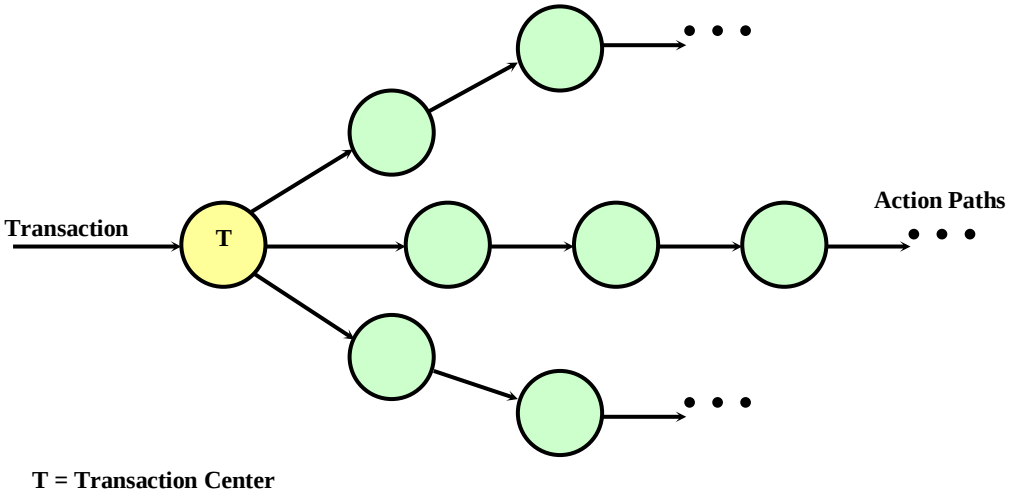


Figure 3.5.6-1: Basic Transaction-Flow Graph

Table 3.5.6-1: Node and Link Definitions for Transaction-Flow Graphs

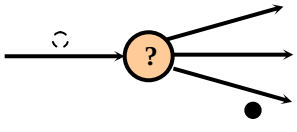
Element	Comments
Origin Node	An entry node of a transaction-flow graph model
Death Node	An exit node of a transaction-flow graph model
Task	Each task in a transaction-flow graph is represented by a node
Branch Node	A node at which an incoming transaction takes one of two or more alternative output paths, designated as a black dot. In this case the incoming transaction has exited on the bottom link. 
Branch Predicate	A predicate that controls which output link of a branch node is taken. The basis for control may be on transaction data values (i.e., via a transaction-control record) or on a combination of transaction type and state.

Table 3.5.6-1: Node and Link Definitions for Transaction-Flow Graphs (continued)

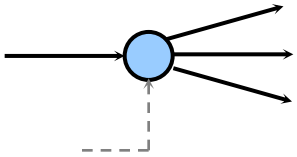
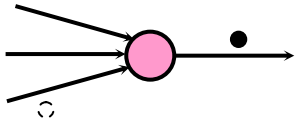
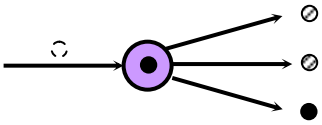
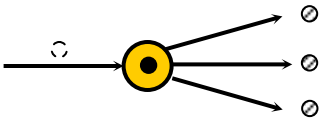
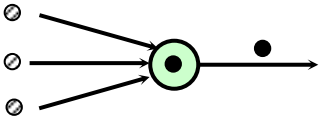
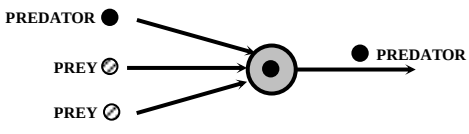
Element	Comments
Control Inlink	An input link that exercises control over which output link of a branch node the transaction record will take. The control value may be independent of the values in the record. There must be a predicate associated with the control input link, which is identified as a dashed line on the transaction-flow graph. 
Junction Node	A transaction entering at any input link of a junction node will emerge at the junction node's output link. This corresponds to junction nodes defined for control-flow graphs. 
Birth Node	A node at which an incoming transaction results in more than one output transaction. These output transactions have individual properties that may, in part, be inherited from the input transaction. 
Split Node	A node at which an incoming transaction results in more than one output transaction, but the original transaction ceases to exist. The output transactions need not be identical, but is assumed to have its own properties, e.g., type and state. 
Merger Node	A node at which two or more incoming transactions merge to create a new output transaction. The incoming transactions cease to exist following the merger. 
Absorption Node	A node with incoming transactions, one of which (predator) absorbs the others (prey). 
Markovian Node	A node whose action (processing, branch, birth, split, etc.) depends only on the type and state of incoming transactions, not on the path by which the transactions reached the node. A Markovian translation-flow graph is one in which all of the nodes are Markovian.

Table 3.5.6-2 describes the basic steps for performing Transaction-Flow testing (adapted from Reference 1).

Table 3.5.6-2: The Transaction-Flow Testing Process

Step	Discussion
1. Verify the specification	Examine and analyze the requirements for completeness and self-consistency. Confirm that the specification correctly reflects the requirements.
2. Identify and name all transactions	All "normal" transactions should be defined within the specification, but there may be any number of implicit transactions that are missing. These might include: • Acknowledgements, receipts, negative acknowledgements • Special installation and checkout transactions • Special operational diagnostics transactions • Transactions that audit other transactions • Transactions used in user- training modes • Initialization or reset transactions for external interfaces • System recovery transactions • System performance measure transactions • System security test transactions • Transactions used for protocols not covered above • Transactions which inquire about status of other transactions • Responses to transaction status requests • Transactions generated by transaction recovery activities • Recovery transactions generated from external systems
3. Define a transaction type hierarchy	Define a hierarchy of transaction types based on all of the explicit and appropriate implicit transactions identified in Step 2. Typically, the same hierarchy used by the developers can be used.
4. Define transaction states	Define the transaction state for each identified transaction type. The states should be defined to correspond to the processing sequence that is associated with the transaction type. If the states are a progression of numbers, then a list should be sufficient. If more complex behavior is anticipated, a better choice for testing may be a finite-state model.
5. Identify transaction characteristics	Identify how each transaction enters the system (origin), leaves the system (death), merges (i.e., with whom), absorbs, splits, gives birth, etc.
6. Define transaction-control records	Define a hypothetical transaction control record for each transaction type that contains, as a minimum, the transaction type and state. For a reasonably high quality transaction processing system, an actual transaction control record implemented in the software can be adopted. For external transactions, an appropriate hypothetical record will need to be defined and developed.
7. Identify all queues	For each queue, define where the transactions come from (the origin), the queue discipline, priority order within the queue discipline, and batch versus continuous processing of the queue. Check capacity limits for all queues that have them. Popular queue disciplines are: • FIFO (first-in, first-out) • LIFO (last-in, first-out) • Batch (all transactions processed when pre-defined conditions are met) • Random (possibly based on a probability distribution for a transaction-control record value) • Priority (fixed or based on a transaction property, with each priority treated as a separately-processed queue) • Multiple server (a server-selection discipline that takes priority over the individual queue discipline)

Table 3.5.6-2: The Transaction-Flow Testing Process (continued)

Step	Discussion
8. Identify processing components	Group the processing components (not necessarily software) according to focus and hierarchical model principles. The transaction-flow model can be applied at various levels of detail, from systems down to code, but recommended focus is at the system level, particularly with respect to correctness of component interfaces; correct component transaction routing; queue management and discipline; mergers; absorptions; splits; births; synchronization; simultaneity; transaction creation/destruction; and transaction duplication/loss.
9. Identify component-specific tests	For each component identified in Step 8, define an appropriate component-level test. These tests might be in the form of a lower-level transaction-flow model, or an entirely different model type.
10. Separate nodes	Separate split/births and mergers/absorptions from their associated processing nodes. This is accomplished by putting an explicit split/birth node after the processing node and an explicit merger/absorption node before the processing node. After completion of Step 10, there should be a set of nodes and links that define the full set of transaction flows that should be tested.
11. Confirm the model	The transaction-flow model should be confirmed by using a model program in a convenient programming language.
12. Define/select test paths	Transaction-flow model test "paths" possess characteristics of both paths (see Topic 3.5.3, Table 3.5.3-4) and slices (see Topic 3.5.5, Table 3.5.5-2).
13. Sensitize the selected test paths	Sensitization is defined as the use of input values that will cause a selected path in the model to be traversed, assuming that there are no defects in the model implementation. The sensitizing procedure is dependent on the nature of the predicates along the path being tested. If the predicates are primarily logical, sensitization is typically performed concurrently with path selection. If the predicates along the path are predominantly numeric or algebraic, then a different process will be used. Sensitize the appropriate paths to be tested by interpreting the input values of the predicates along the path. These interpreted predicates result in a set of conditions and mathematical inequalities that will provide a solution set such that specific solution in that set will cause the selected path to be traversed.
14. Predict/record expected outcome for each test	See Topic 3.5.3, Table 3.5.3-4 and Topic 3.5.5, Table 3.5.5-2
15. Define the validation criteria for each test	Before testing commences, define and document what the validation criteria will be for each test performed, i.e., what are the outcome results that will be acceptable as an indicator that the test has been successfully passed (pass/fail criteria). Define how normal and failure case scenarios will be handled.
16. Perform the tests	Automate, automate, automate
17. Confirm each test result	Compare test results with those predicted, and with the defined validation criteria, to determine whether a test has passed or failed. For failed tests, proceed with root-cause analysis and corrective action identification, implementation and verification. For tests that have passed, go to Step 18….
18. Verify the path	Path verification is needed to avoid the pitfalls of potential "coincidental correctness". It is not necessary to verify every computation, but as much of the path should be verified as is convenient given the available resources and constraints. Another guard against coincidental correctness is to test several cases along each defined path. This happens as a natural result of other test techniques, particularly domain testing.

As with Data-Flow Testing (Topic 3.5.5), there is a range of progressively more powerful test method coverage criteria that can be used on transaction-flow graphs. These are summarized in Table 3.5.6-3 (adapted from Reference 1).

Table 3.5.6-3: Hierarchy of Transaction-Flow Graph Test Methods

Test Method	Comments
Origin/Exit, Birth/Death Cover	Run a sufficient number of tests to ensure that every transaction origin and birth has been exercised, and that all intermediate and outgoing transactions have been produced. Include all transaction types in the test
Node Cover	Node cover alone is also insufficient, since it only reaffirms what should have been tested in a lower-level model. It is better than nothing, however, as it at least confirms that all births, splits, mergers, absorption and queue disciplines are working correctly.
Link Cover	Link cover testing not only confirms the correctness of individual nodes, but also how they work with each other. System testing does not exist (at least it should not) without link cover. Link cover confirms the same correctness that node cover does, but it also ensures that the right transactions are processed at every step of the model.
Slices	The concept of a "slice" for transaction-flow testing is almost identical to that of data-flow testing. If there are only branch and junction nodes within the model, then a slice corresponds to an entry/exit path. If the model contains births or splits, a slice is constructed by following all of the output links of the birth or split node to their death nodes. If the model contains merger or absorption nodes, a slice is constructed by following the input links back to the pints at which the merged (predictor/prey) transactions were born, or were introduced into the system.

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.7: Domain Testing

Domain testing represents a formal technique that can be automated to replace the historically used practice of testing extreme input values and their combinations. It is based on the formal definition of processing domains as sets of mathematically-stated inequalities that are defined over the required input space. It is not feasible to test an entire system, or even an entire program, using domain testing. Some of the main characteristics that may indicate a suitable application of domain testing are:

- Segments of specifications that are explicitly stated in terms of algebraic inequalities
- Extensive numerical processing using a great deal of conditional logic
- Inputs that are numeric by nature and require thorough data validation and characterization, even if subsequent processing is not predominantly numeric-based
- Systems that may or may not contain software, if they can be described (at least in part) in terms of algebraic inequalities

Table 3.5.7-1, adapted from Reference 1, provides an overview of some of the more important terminology specifically associated with domain testing.

Table 3.5.7-1: Important Domain Testing Terminology

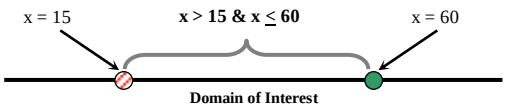
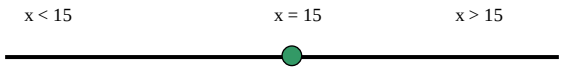
Terminology	Comments
Domain	A subset of the input space over which any processing performed by a system being tested is defined. Domain testing determines whether a specific set of input values (i.e., an input vector) is within that domain or not. Domains are defined by a set of mathematical boundary inequalities.
Domain Boundary	The means by which a domain is defined, which typically take the form of an algebraic inequality expression.
Domain Boundary Set	A set of inequalities that, when taken together, define the valid region of a domain. 
Boundary Inequality	An algebraic expression over the input variables that, in part, defines which points in the input space belong to the specific domain of interest. As an example (refer to the graphic below), an inequality expression x ≤ 15 defines the domain of interest as having all values less than or equal to 15. 
Boundary Equation	The equation obtained by converting a boundary inequality into an equation. For an example, a boundary inequality may be stated as "x ≤ 6" ('x' is less than or equal to 6). Conversion to a boundary equation results in "x = 6".

Table 3.5.7-1: Important Domain Testing Terminology (continued)

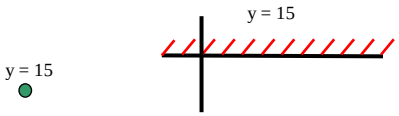
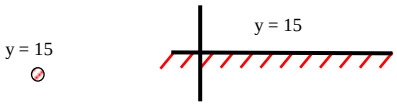
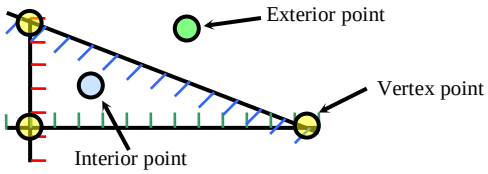
Terminology	Comments
Closed Boundary/Domain	A boundary of a domain is closed if the points on the boundary are included in the domain of interest. For "y ≥ 15", the value of 15 is included in the domain, hence the boundary is closed. A closed domain is one in which all boundaries are closed. For a one-dimensional representation, closed boundaries may be indicated by a solid dot. In two dimensions, closed domains may be indicated by hash-marks on the closed side of the domain. For "y ≥ 15" 
Open Boundary/Domain	A boundary of a domain is open if the points on the boundary are not included in the domain of interest. As an example, for "y > 15", the value of 15 is not included in the domain, hence the boundary is open. An open domain is one in which all boundaries are open. Domains need not be either all open or all closed, i.e., it can have at least one open and/or at least one closed boundary. For a one-dimensional representation, an open boundary may be indicated by a hash-marked dot. For two dimensions, there are no hash-marks indicated on the open side of the domain. For "y > 15" 
Points	A **vertex** point is one through which two or more boundaries cross. For "n" dimensions, a vertex point is the solution to "n" simultaneous, linearly independent boundary equations. An **interior** point is one that lies within a domain of interest. An **exterior** point is one that lies outside the domain of interest.  An **ON** point is one that is on the domain boundary, or is as close to the boundary as possible while still satisfying the defined boundary conditions. An **OFF** point is an *interior* point with respect to a boundary if the domain is open, or an *exterior* point just outside the boundary if the domain is closed. An OFF point does not satisfy the conditions associated with a boundary.

Table 3.5.7-1: Important Domain Testing Terminology (continued)

Terminology	Comments
Degeneracy	A degenerate domain, for "n" dimensions, is a domain having less than "n" dimensions. In two dimensions, a degenerate domain consists of either a point or a line. In three dimensions, a degenerate domain consists of a point, a line, or a plane. A degenerate boundary, for "n" dimensions, is a domain boundary of less than "n-1" dimensions. In two dimensions, a degenerate domain boundary consists solely of a point. In three dimensions, a degenerate domain boundary consists of a line or a point, rather than a plane.
Completeness	A **complete boundary** extends to $\pm \infty$ in all of its variables. A **boundary segment** is defined as part of boundary inequality between two or more domains, i.e., it is one of the edges of a domain. An **incomplete boundary** is a boundary with one or more gaps. Gaps, if they occur, are between vertex points, i.e., they consist of boundary segments.  Complete Boundary  Incomplete Boundary
Closure	**Consistent closure** is defined as a boundary for which the closure direction (open versus closed) is the same along its entire length. **Inconsistent closure** is defined as a boundary condition for which the closure direction changes at least once along its entire length. Closure changes, if and when they occur, typically occur at vertex points (i.e., between boundary segments).  Consistent Closure  Inconsistent Closure

Table 3.5.7-2, derived from Reference 1, provides an overview of the hierarchical approaches that should be used in domain testing to ensure good node and link coverage. Nodes (or objects) are those domains defined over the input vector. Links (or relations) are defined as "*is adjacent to*". In general, the direction of the inequality defines the direction of the link. For two adjacent domains, and assuming that Domain 1 is closed, an arrow would be drawn from Domain 1 to Domain 2.

Table 3.5.7-3, also derived from Reference 1, identifies a variety of domain testing techniques that should (or should not) be considered.

Table 3.5.7-2: The Hierarchy of Node and Link Coverage

Coverage Area	Hierarchy
Node (or Object)	1. Test at least one point in each domain to confirm that the correct processing has been selected and, if selected, executed correctly by the CASE statement. 2. For domains consisting of sub-domains (whether adjacent or not), confirm that all of the required pieces are present, and that they receive the required processing. 3. Verify that there are no overlapping domains. This can be accomplished by using graphs for one- and two-dimensions, or higher-order algebraic techniques for more than two dimensions. Typically, domain overlaps occur on boundaries. 4. Confirm that the input space is complete. Every input vector must be handled, even if it results in rejection of the input. Inspection can be used to handle one- and two-dimensional cases, but algebraic techniques must be used for more than two dimensions
Link (or Relation)	1. Confirm that all domains that are considered to be adjacent are, in fact, adjacent. Adjacent boundaries must have a boundary inequality between them. 2. Confirm that extra boundaries do not exist. 3. Confirm the correctness and accuracy of each boundary inequality.

Table 3.5.7-3: Strategies for Performing Domain Testing

Strategy	Comments
Test Extreme Points (Heuristic)	Also referred to as "boundary value testing", "extreme value testing" and "special value testing". The strategy recommends testing any numerical input at and near the allowed minimum and maximum values for that input. Reference 1 states that formal domain testing should be considered over heuristic domain testing, as the former will do better testing and find more defects using fewer tests.
Test Extreme Point Combinations (Heuristic)	Popular (but, according to Reference 1, misguided) strategy that tests the combinations of extreme points. It assumes that there is an upper and lower acceptable value for every input variable. The process generates many tests (for "n" input variables, 4^n + 1 tests are generated), most of which may be meaningless at best, or misleading at worst.
Weak 1 x 1, One Dimension (Formal)	Test is "weak" from the standpoint that it only does one set of tests for each boundary inequality instead of one set of tests for every boundary segment. The "1 x 1" nomenclature indicates that it will require one "ON" point and one "OFF" point for each boundary inequality. Possible defects detected may be (1) closure defects (boundary is opposite of what it should be – open or closed), (2) left-shifted boundary (OFF point gets the wrong processing), (3) right-shifted boundary (ON point gets the wrong processing), (4) missing boundary (ON and OFF points both get the same, but wrong, processing, and (5) extra boundary (extra boundary divides one original domain into two domains).

Table 3.5.7-3: Strategies for Performing Domain Testing (continued)

Strategy	Comments
Weak 1 x 1, Two and Higher Dimensions (Formal)	Test is "weak" in that it assumes that every boundary extends to $\pm \infty$, there are no gaps in the boundary, and closure is consistent along the entire boundary length. First task is to determine if there is a defect associated with a boundary. Boundaries with no defects don't require additional testing. A stronger strategy is needed to determine what's wrong with the boundary as implemented.
Weak N x 1, "N" Dimensions (Formal)	A higher-order strategy that can be used to (1) ensure that various kinds of domain defects won't escape the basic "1 x 1" tests, (2) learn something about the defects that have been identified, and (3) provide insights into the general testing strategy. All higher-order test strategies require significantly more test points. By selecting "n" ON points, the correctness of an "n-1" dimensional boundary hyperplane in "n" dimensions can be confirmed. In addition to closure defects and extra/missing boundaries, other potential defect situations are (1) up or down domain shift (equivalent to a one-dimensional left- or right-shift) and (2) domain tilt (no one-dimensional equivalent), which represents any error in a coefficient of the inequality being tested.
Strong Domain Testing	Where weak domain testing only exercises one set of tests for each boundary inequality, strong domain testing exercises a separate set of tests for each boundary segment. Reasons to use strong domain testing include (1) the existence of gaps in a boundary inequality (at least three segments that need to be tested), (2) a closure change in one or more boundary segments, (3) processing such that some test points will fall into an unprocessed region and be rejected, and (4) ad-hoc, disorganized software coding styles.

For More Information:

1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.8: Finite-State Testing

One of the most fundamental models for software testing is the finite-state model which is structured around the definition of a state-transition table that includes every possible state-input combination. Finite-state models have their roots in hardware logic testing. This category of model is considered excellent for testing menu-driven applications and is used extensively in applications based on object-oriented designs.

Table 3.5.8-1, derived from Reference 1, presents the terminology that defines the elements of finite-state testing.

Table 3.5.8-1: Elements of Finite-State Testing

Basic Element	Terminology
Input	**Input event:** A distinct event that is repeatable, or a fixed sequence of events that are characterized by inputs (or an input sequence) to a system. Inputs are based on their ability to control the system (i.e., change states) rather than on data. An example is an input whose value causes the software to follow a different processing path. **Input encoding:** Each input event can be assigned a number or a name, thereby encoding the events onto integers or a set of characters. The behavior of a finite-state model will not be changed by modifying the input encoding scheme. **Input symbols:** Symbols represent the names or values associated with the input encoding. **Number of input symbols:** Input encoding can assign an integer to every distinct input event, presumably with no gaps within the numbers. The number of input symbols should typically be small (not greater than 20) to make manual analysis of the model reasonable. More than 20 input symbols is an indication that tools will be needed to analyze the model. **Reset:** A special type of input that forces a transition to an initial state from any other state. A reset capability is not essential, but makes test design and testing much easier.
State	**State:** In a finite-state model, each state is graphically depicted by a node **State encoding:** States can also be numbered (**state code**). Each numbered state should have an operational meeting. States of a disk drive may include (1) start-up; (2) motor on; (3) seek track; (4) seek sector; (5) reading; (6) writing; (7) erasing; (8) motor off. The behavior of a finite-state model will not be changed by modifying the state encoding scheme. **Current state:** At a discrete point in time, a system can only be in one state, the current state. **Initial state:** The state of a system prior to any input being received. **State counter:** A hypothetical or actual memory location that holds the state code of the current state. A state counter can be explicit or implicit. **Number of states:** The state counter has a maximum value, typically under 30 (assuming no gaps in the state numbers). Tools are usually required to handle more than 30 states. **Finite-state machine:** An abstract machine representation for which the number of states and input symbols are fixed and finite. A finite-state machine is comprised of **states** (nodes), **transitions** (links), inputs (link weights) and outputs (link weights). Finite-state machines are commonly depicted by **state graphs** (see Figure 3.5.8-1) **Reachable state:** One state is considered reachable from another state if there is a sequence of inputs such that, when starting from the originating state the finite-state model will end up in the subsequent state. The state graph will have a link between the two states. **Unreachable state:** A state is unreachable from other states, especially from the initial state, if a correctly drawn state graph contains no links to that state. An unreachable state usually means a defect is present.
State (continued)	**Isolated states:** Isolated states are a set of states that are not reachable from the initial state. Within this set, the states may or may not be strongly connected (i.e., may or may not contain defects), but the fact that they are unreachable from the initial state is what defines them as isolated. Isolated states should, as a minimum, be considered suspicious for the purposes of testing. **Initial state set:** A set of states that include the initial state(s), where the initial state(s) may or may not be strongly connected. Once a transition occurs outside the initial state set, the finite-state model cannot return to the set (e.g., booting up a computer). **Working states:** Once the system leaves the initial state set, a set of strongly connected working states is reached. Finite-state testing focuses mostly on the set of working states. **Working set initial state:** The first state in the set of working states (e.g., the first menu that appears once a computer system has booted up). That is, the first state that can be tested within the set of working states. **Exit state set:** A finite-state model may have one or more states, or set of states, for which there is no path back to the working set once the exit state set has been entered (e.g., the exit sequence for a program).

Table 3.5.8-1: Elements of Finite-State Testing (continued)

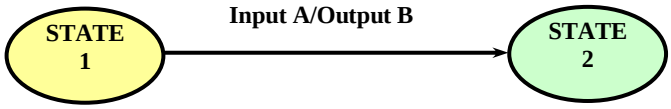
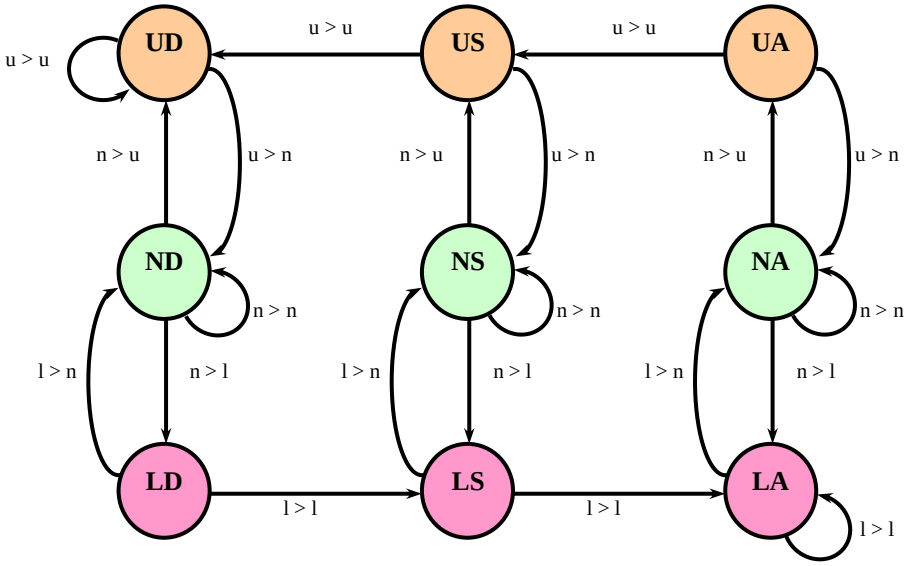
Basic Element	Terminology
Transition	**Transition:** As a system responds to an input event, it may change states (**state transition**). Transitions are denoted by links between state nodes, as indicated below, where an input event (A) has triggered a system transition (link) from State 1 to State 2:  **Self-transition:** This represents a link from a state back onto itself. There may be an output associated with that transition. **Initial state tour:** A sequence of transitions from one state (say, from the initial state in the initial state set or from an initial state in the working set) to a second state, and back to the original state.
Output	**Output encoding:** There may be an output action associated with a state transition. Output actions can also be mapped onto integers or character strings. System behavior is not affected by any modifications to the output encoding scheme. **Output event:** As a result of a state change, a system may produce an output, equivalent to outputting an integer such as the output code. For example, an output of "5" may correspond to selection of the fifth item on a pull-down menu. Outputs, as with inputs, are denoted as a link weight. In the example above, Output B is activated by Input A. **Null output:** A hypothetical output event, e.g., do nothing in response to an input event.

Figure 3.5.8-1 provides an over-simplified representation of a state graph for an elevator. The elevator has three possible states, i.e., it will either be stopped (S), ascending (A), or descending (D). The inputs that control the movement of the elevator can be described as neutral (N – the elevator completes its current activity before responding to a pushed button), going up (U – somebody has pushed a button requesting the elevator to go UP), and going down (L for "lower" - somebody has pushed a button requesting the elevator to go DOWN). Complicating factors, such as whether the doors are opening or closing and how many floors the elevator is going up or down, have been ignored. This finite-state model example requires that the elevator must always go through the neutral phase before it will go up or down, i.e., it won't immediately change direction or move once the UP or DOWN button is pushed. The figure also includes the state table (or state-transition table) corresponding to the state graph ("impossible" combinations are shaded).



STATE	u>u	u>n	n>n	n>u	n>l	l>l	l>n	u>l	l>u
UD	UD	ND							
US	UD	NS							
UA	US	NA							
ND			ND	UD	LD				
NS			NS	US	LS				
NA			NA	UA	LA				
LD						LS	ND		
LS						LA	NS		
LA						LA	NA		

Figure 3.5.8-1: Example State Graph with Supporting State Table

Common software applications that should be considered as candidates for finite-state testing include (from Reference 1):

- Menu-driven software
- Object-oriented software
- Protocols
- Device drivers
- Former hardware
- Installation software
- Backup/recovery software

Table 3.5.8-2 provides an overview of the finite-state testing process.

Table 3.5.8-3 describes those things that should be verified as part of a thorough finite-state test scenario.

Table 3.5.8-2: The Finite-State Testing Process (Reference 1)

Step	Discussion
1. Identify inputs	Identify the specific input events that are going to be modeled, and give each event a unique name and characterize it. Not every input possibility should be included in the model. It is preferred to keep the specific number of events to be modeled between 10 and 20.
2. Define input encoding	Input codes developed by test personnel may or may not match those used by the original programmer. It is not necessary to encode any inputs that are not going to be tested. The input encoding *process* should be tested, however, if it is part of a program's implementation.
3. Identify states	States are frequently created as a product of factors. If so, the factors should be identified, realizing that there should be a unique state for every combination of factors. List all of the states identified and define them using a logical set of names.
4. Define state encoding	If the software designers used an implementation that reflects a finite-state machine (see Reference 2), there may already be an existing state encoding process and counter. In this case, the state encoding process should be tested. If a finite-state design doesn't exist, then state encoding becomes part of the finite-state model development (not necessarily part of the software). State encoding correctness still needs to be verified.
5. Identify output events	Output events are more likely to consist of a sequence of actions, rather than simple, single output events. Each sequence of actions should be identified and given a name.
6. Define output encoding	If the design represents an explicit finite-state design, the programmers may already have an output encoding that can be tested. Output encoding must be tested to verify that it corresponds to what actually happens.
7. Build/clean up state and output tables	This represents the most difficult, time consuming and error prone part of the finite-state testing process.
8. Design tests	There are three kinds of test that need to be designed: (1) input encoding verification tests, (2) output encoding verification tests, and (3) state/transition verification tests. The first two tests are tests of the finite-state model. The last test covers both the software and the finite-state model.
9. Run tests	Each defined test should begin from the initial state, make a tour to get to another state, and then return to the original initial state.
10. For every input, confirm its transition and output	Not a trivial task, it typically requires some level of design support. Each test starts from the initial state and takes the shortest untested tour back to the initial state, i.e., each test builds on previous, simpler tests. Tests are added to ensure sufficient link coverage. Once a set of covering tours have been defined, the input code needed for each transition and the output code associated with each transition are also defined. Proceed backward through the input encoding to find actual inputs, and forward through the output encoding to find the relevant details of the output action.

Table 3.5.8-3: Finite-State Testing Verification (Reference 1)

What to Verify	Why
Input encoding	The model should match the input encoding that was implemented. If not, either the model or the input encoding is incorrect. If inputs are numerical, domain testing could be used for verification. If inputs are character strings, syntax test (see Reference 1) or another state model can be used.
Output encoding	There should be relatively few distinct output events or action sequences, each with a defined name. Proper action sequences should be verified with actual performance. In the event of problems, examination of intermediate computation steps and/or file activity logs, or use of a symbolic debugger, may have to be used.
Initial state	It is assumed that there is at least one initial state and a path to get there. Complicated systems may have a number of initial states. Having the software indicate that it is in an initial state may be insufficient if that is the defect that the test is trying to capture. Verify all of the properties of each initial state tested, such as which files are open, resource use, active programs, etc.
Exit state	There are typically several potential exit states or a set of exit states. Verification is needed that every exit state can be reached (both in the model and as part of the test).
Verify state	All of the states that should exist should be verified as present, which is the purpose of doing a tour to a target state. If there is an explicit state counter and the software designers have built in suitable testability, then there is a means for knowing the software state at any given time. Testing becomes more difficult if there is limited or no means to identify a state.
Extra states	If they exist, there are usually a large number of extra states ("parallel universes") in software. Extra states result from hidden finite-state behavior. Although the defects in many of these states may be harmless, there may be specific universes where the defect is fatal to the system. Reference 1 has a more detailed explanation on extra states.
Confirm every transition	There is a potential output (or outcome) associated with each transition. Each transition is comprised of a new state and any outputs that may be generated. Null outputs must also be confirmed (verify that nothing happens). Every component of the state encoding should be confirmed.

For More Information:

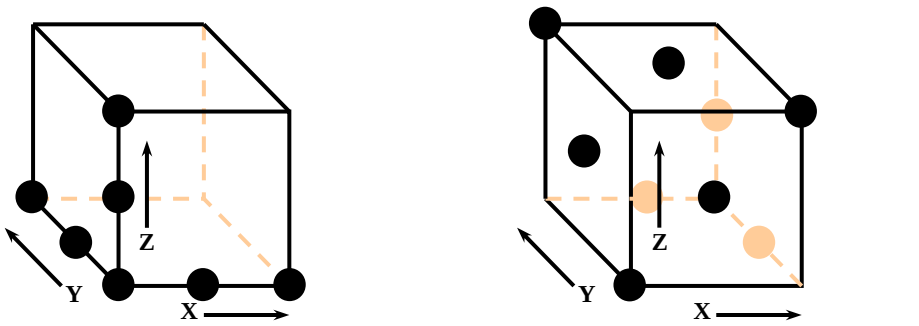
1. Beizer, B., "Black-Box Testing: Techniques for Functional Testing of Software and Systems", John Wiley & Sons, May 1995, ISBN 0471120944
2. Beizer, B., "Software Testing Techniques", The Coriolis Group, June 1990, ISBN 1850328803
3. http://www.itl.nist.gov/div897/ctg/stat/mar98ir.pdf
4. http://www.csis.hku.hk/~tse/Papers/xqw.pdf

Topic 3.5.9: Orthogonal Array Testing

Orthogonal array testing is a statistical black-box testing technique that enables the design of a reasonably small set of test cases when the prospect of exhaustive testing becomes impractical or impossible. The purpose of orthogonal array testing is to assist in the selection of appropriate combinations of factors to provide maximum test coverage from using a minimum number of test cases. It is particularly useful for focusing on categories of faulty logic likely to be present in a software component (without examining the code), commonly described as region faults. The basic premise of orthogonal array testing is that system functionality can be defined using parameters, and that these parameters can be represented by ranges of values, or that parameter values are discrete elements of a finite set.

Orthogonal array testing selects test cases in a manner that exercises the interactions between independent measures or parameters, defined as ***factors***. Each factor is defined within a finite set of possible values, defined as ***levels***. In the tabular representation of an orthogonal array, each column in the array corresponds to a factor and each row corresponds to a test case. The test cases are created to define all possible pairwise combinations of levels for the factors.

The conceptual difference between conventional testing and orthogonal array testing is illustrated in Figure 3.5.9-1. Conventional testing takes an approach that considers only one input at a time, testing levels (or values) across each factor, with all other factor values held constant (Figure 3.5.9-1a). Orthogonal array testing develops test cases that provide more complete coverage across the test domain by dispersing tests uniformly (Figure 3.5.9-1b). Each dot in the figure represents a test case. The set of test cases for this example of orthogonal array testing is one in which only two-way interactions are covered, rather than higher-order interactions. No pair of values appears more than once in the array. The conventional approach, given three separate factors (X, Y, Z), each of which is tested at three separate values (e.g., 1, 2, 3) would require $3^3 = 27$ tests to cover all combinations. The pairwise orthogonal array can cover these combinations with 9 tests, representing a significant improvement in testing efficiency.



(a) Conventional – single input (b) Orthogonal array – 2-way interactions

Figure 3.5.9-1: Conventional Versus Orthogonal Array Testing

The number of test cases required to cover all combinations of factors/levels is given by the generic formula:

$$\text{No. of Test Cases} = (\text{No. of Values per Factor})^{\text{No. of Factors}}$$

L_4 (2^3)

	FACTOR		
CASE	A	B	C
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

L_8 (2^7)

	FACTOR						
CASE	A	B	C	D	E	F	G
1	1	1	1	1	1	1	1
2	1	1	1	2	2	2	2
3	1	2	2	1	1	2	2
4	1	2	2	2	2	1	1
5	2	1	2	1	2	1	2
6	2	1	2	2	1	2	1
7	2	2	1	1	2	2	1
8	2	2	1	2	1	1	2

L_{16} (4^3)

	FACTOR		
CASE	A	B	C
1	1	1	1
2	1	2	2
3	1	3	3
4	1	4	4
5	2	1	4
6	2	2	1
7	2	3	2
8	2	4	3
9	3	1	3
10	3	2	4
11	3	3	1
12	3	4	2
13	4	1	2
14	4	2	3
15	4	3	4
16	4	4	1

L_9 (3^4)

	FACTOR			
CASE	A	B	C	D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Figure 3.5.9-2: Example Orthogonal Array Structures

Figure 3.5.9-2 provides tables of orthogonal arrays for an L_4 array (where "4" equals the number of test cases based on 3 factors, each with 2 unique levels), an L_8 array (8 test cases, 7 factors, 2 levels), an L_9 array (9 test cases, 4 factors, 3 levels), and an L_{16} array (16 test cases, 3 factors, 4 levels).

The initial development of a suitable orthogonal array begins with the mapping of an array to a specific problem, i.e., mapping quantitative numbers to software program functionality. Consider an example using an L_9 array based on four factors, each with three possible values. The four factors are:

Factor A: Operating System
Factor B: RAM Size (in MB)
Factor C: Database Size (Number of Records)
Factor D: Processor Speed (in MHz)

The possible "values" for each of these factors (with its corresponding orthogonal array value) is presented in Table 3.5.9-1.

Table 3.5.9-1: Orthogonal Array Mapping

Orthogonal Array Value	Factor A (Operating System)	Factor B (RAM Size)	Factor C (Database Size)	Factor D (Processor Speed)
1	Windows 98	128 MB	100	800 MHz
2	Windows ME	256 MB	1000	1000 MHz
3	Windows NT	512 MB	10000	1500 MHz

Mapping the L_9 orthogonal array presented in Figure 3.5.9-2 to this example yields the nine test cases presented in Table 3.5.9-2.

Table 3.5.9-2: Mapping an L_9 Array to Example

Test Case	Factor A	Factor B	Factor C	Factor D	OS	RAM	Size	Speed
1	1	1	1	1	W98	128	100	800
2	1	2	2	2	W98	256	1000	1000
3	1	3	3	3	W98	512	10000	1500
4	2	1	2	3	WME	128	1000	1500
5	2	2	3	1	WME	256	10000	800
6	2	3	1	2	WME	512	100	1000
7	3	1	3	2	WNT	128	10000	1000
8	3	2	1	3	WNT	256	100	1500
9	3	3	2	1	WNT	512	1000	800

To assess test results, the orthogonal array should be used in the following manner:

1. Detect and isolate all single-mode faults (a specific value of one factor consistently causes an error condition)
2. Detect all double mode faults (highlights a consistent problem when specific values for two parameters occur together, such as pairwise incompatibility or harmful interactions)
3. Multimode faults can also be detected using orthogonal array test strategies, but the arrays are typically more complex than those presented in this section and should be examined only after the first two fault categories are evaluated

There are a number of commercially available tools for creating orthogonal test arrays. The AETG tool offered through Telcordia Technologies represents one of these. An Internet link to a technical paper that describes this tool in more detail is provided below.

For More Information:

1. Phadke, M.S., "Quality Engineering Using Robust Design", Prentice-Hall, 1989, ISBN 0137451679
2. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783
3. Roy, R., "A Primer on the Taguchi Method", Van Nostrand Reinhold, 1990, ISBN 0442237294

Topic 3.5.10: Software Statistical Usage Testing

Statistical usage testing (SUT) represents the application of statistical sampling theory to software testing and certification of reliability as a critical element of the Cleanroom approach to software development. In statistical testing, software testing is perceived as a problem to be solved using proven statistical methods. The basic premise underlying the use of SUT is that the ability to test all possible ways in which software might be used is necessarily constrained by severe cost (dollars) and resource (time) limitations. As an example (from Reference 2), if a (overly simplified) system has (1) between one and 10 inputs, (2) 20 different inputs are possible, and (3) inputs may be repeated, then the total number of usage scenarios is represented by:

$$\sum_{n=1}^{10} 20^n = 10,778,947,368,420 \text{ possible test cases}$$

At 1000 test cases per second on each of 100 copies of the software, testing all possible test cases would require > 3.4 years.

In today's marketplace, months between software releases, not years, are the measure of product success.

Whereas traditional software testing has focused on branch coverage, path coverage, and boundary value testing as a means to uncover defects in the software, the focus of SUT is on how the software fulfills its intended purpose from the users' perspective. The primary objectives of SUT are two-fold:

Find Faults Having the Most Influence on Reliability, Security, and Safety from the Users Perspective

The focus of SUT, particularly for the DoD and other security- or safety-critical industries, must include test cases that will detect failures whose occurrence may otherwise compromise security; cause personnel injury or death; cause equipment damage or destruction; or impair ability to complete a critical mission.

Produce Data That Makes Possible the Certification and Prediction of Software Reliability

The ability to certify and predict software reliability provides a means to know when testing can be stopped and the product accepted. This premise implies that a usage profile must be defined in order to support an appropriate rationale for certification.

The success of statistical usage testing, then, hinges on the ability to accurately characterize the population of possible software uses, and how the subset of test cases to be applied will be determined.

Figure 3.5.10-1 provides an overview of the Statistical Usage Testing process. Tables 3.5.10-1 through 3.5.10-6 present, in more detail, the elements that comprise each phase of the process. Relevant definitions to be considered as part of the SUT process include:

- **Statistical Testing:** The application of statistical science to the testing of software intensive systems
- **Usage Model:** The characterization of all possible uses of the software at a pre-defined level of abstraction, preferably constructed before any code is written so that testability enhancements can be factored into the software specification
- **Test Case:** The transverse of any single, unique path of the usage model based on either random or crafted (non-random) test cases from a usage probability distribution

- **Certification:** The attainment of reliability and statistical confidence goals for a specifically defined usage environment based upon a well-defined and repeatable test protocol that can be evaluated without bias

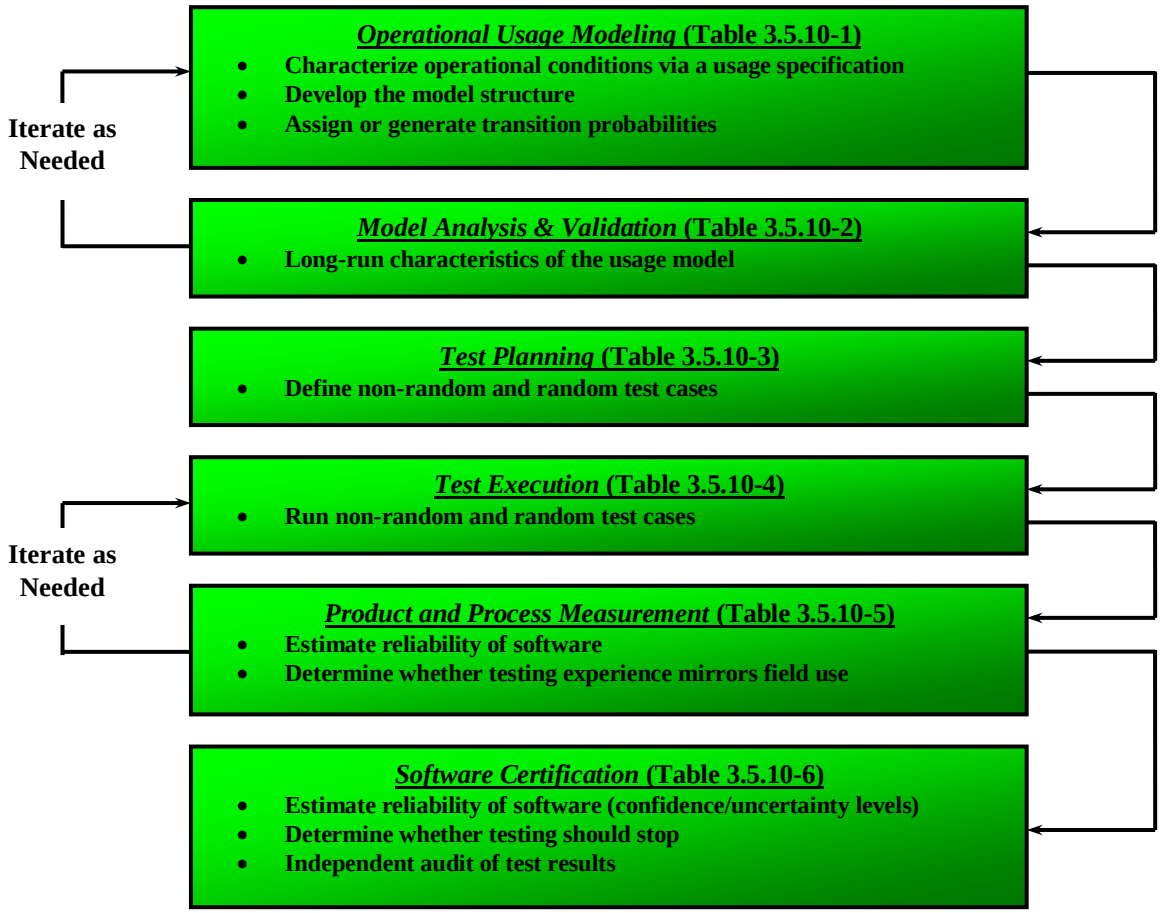


Figure 3.5.10-1: Overview of the Statistical Usage Testing Process

Table 3.5.10-1: Operational Usage Modeling

Characteristics
• Some form of specification that describes the correct behavior of the software is needed before model development can be initiated. This behavior can be defined either within a formal specification, supporting requirements documentation, a user's manual, or a predecessor system. • Different types of usage models include: o Tree-structure models, which assign probabilities to event sequences (Reference 1 and Topics 3.3.1 and 3.5.11) o Markov-based models, which can specify more complex usage and model single events (References 2 and 3 and Topic 3.5.12) • The primary purpose of a usage specification is to provide a basis for how to select test cases for SUT • The operational environment is defined by the software user, the software use, and the environment in which the software will operate. o A ***user*** may be a person, a hardware device, or other software. Sub-classifications of users can also be made, e.g., a person may be further classified by job type, access privileges, or domain experience. o The ***use*** of the software may be represented by a work session, a transaction, or any other unit of service bounded by appropriate start/finish events o The ***environment*** can be characterized by a usage platform, the number of users, the number of concurrent applications, system loading characteristics, etc. • The structure of the usage model may be represented by a graph, in which graph nodes represent usage states and arcs between usage states represent any stimuli that will cause transitions between states (see Topic 3.5.12). The usage model may also be represented as tables or matrices, with rows and columns representing states, and each cell representing the probability of that specific row state being followed by its specific column state (see Topic 3.5.11). • The structure of the usage model represents the ***possible*** uses of the software. A probability distribution needs to be imposed on the model to represent the ***expected*** software use under specified conditions. These probabilities may be assigned based on field data, interviews with customers, or instrumentation of previous software versions. In the absence of such data, probabilities can be generated as a solutions to a set of equations or inequalities that represent constraints on system performance, as given below: o Structural constraints explicitly defined by the model that define the allowed states and the possible or impossible transitions between them o Usage constraints that represent known or expected patterns of system use o Management constraints that impose controls on the testing process to enforce budget (only "x" dollars allowed for testing), schedule (only "y" hours allowed for testing) or policy decisions (software released when reliability achieves level "z") • The Engineering Practice steps (Reference 2) for Operational Usage Modeling are: 1. Identify the system boundary; all hardware, software and human users of the software; and all stimuli that they can send the software 2. Define the usage model structure as it relates to possible sequences of input stimuli, identifying those areas where the usage specification may result in excessive (i.e., nonessential) system complexity and cost. Make recommendations for appropriate simplification. 3. Define the most important software usage environments, such as routine use, safety-critical use, fully loaded use, etc., and determine an appropriate number of environments to model. Repeat this step for each defined model. 4. Define the corresponding probabilities of the usage model

Table 3.5.10-2: Model Analysis and Validation

Characteristics
• Analytical results from the model are calculated to verify that the model accurately represents (or at least reasonably represents) the expected use of the software, after which the model can be used as an effective test planning tool. • Standard calculations from Markov chain-based models provide expected values for several measures that are useful in test planning, resource allocation, safety analysis and field support, including: o Average number of events, or state transitions, per random "use" (test case) o Long-run occupancy of each state (usage profile as a percentage of total usage time spent in each state) o Average number of uses (test cases) before a specific usage state occurs (first occurrence) o Probability of occurrence of each state in a random use of the software (occurrence probability) o Expected number of occurrences of each state in a random use of the software (occurrence frequency) o The average length of a use case or test case (expected sequence length) • The above results, available before the commencement of software design and implementation, can be applied to all phases of the software life cycle, e.g., they can be used to "simplify" the specification, assess software complexity, provide focus to software verification needs, identify event/transition frequencies, enhance test schedule planning, and determine boundaries on inferences concerning software reliability • Reference 3 presents new analytical results for Markov chain models as: o A means for calculating the expected number of test cases need to cover a defined state or transition arc *with associated variance* o A lower bound on the expected number of test cases needed to cover all defined states or transition arcs *with associated variance* o A corrected calculation for the probability of a transition arc appearing in a test case • The Engineering Practice steps (Reference 2) for Model Analysis and Validation are: 1. Generate the standard analytical results for the model, interpreting analytical values in terms of the specification and expected usage to validate their correctness or reasonableness 2. Change the model structure or constraints as needed. Changes to model structure may be needed to more accurately reflect the specification. Changes to constraints may be needed to more accurately represent usage or test management issues. 3. Repeat steps 1 and 2 if the usage model has been changed 4. Generate some test cases and confirm that the results appear to be realistic. If not, return to step 2. 5. Use the model, and the implications derived from it, to support development of performance planning, correctness verification, safety analysis, test planning and reliability improvement activities

Table 3.5.10-3: Test Planning

Characteristics
• As indicated in Table 3.5.10-2, there are several measures that come out of the model analysis and validation activities that support test planning. After the usage model has been developed, test cases can be developed, either manually or automatically, by traversing the usage nodes or states of the model, guided by the transition probabilities associated with exiting each node or state. This traversal results in an accumulation of successive stimuli (i.e., inputs) that represent a unique test case, where the set of defined test cases constitute a *script* to be used to conduct the testing. • Test cases, and the test script, may be annotated during test planning to record results and observations. • Non-random (or crafted) test cases can remove uncertainty about how a system may perform under various circumstances and can contribute to effectiveness and control over all testing. Non-random test cases are generated from the usage model and can include hand-crafted tests, test cases prioritized by probability of occurrence, and test cases generated to cover all usage model transitions in the minimum number of steps.

Table 3.5.10-3: Test Planning (continued)

Characteristics

• Examples of non-random testing that may be useful prior to random testing include:
  ○ Model coverage tests, where an algorithm representing the model structure is used to generate the minimal sequence of test events (least cost sequence) to cover all arcs/states
  ○ Mandatory tests representing specific test sequences that may be required to satisfy contractual, policy, moral or ethical issues
  ○ (Nonrandom) regression tests that can help uncover cost-ineffective redundancies in the test suite and assess the impact of omitting them
  ○ Critical but likely use tests that represent critical states, transitions and subpaths that may have a low likelihood of occurrence in field use (or in a random sample), but whose failure may represent catastrophic safety-critical consequences to personnel, equipment, or mission success
  ○ Importance tests (sampling) that add management constraints and an objective function that produces transition probabilities that emphasize the "value" in the sampling process
  ○ Partition tests that can increase sampling efficiency
• Random test cases are, as the name implies, randomly generated from the usage model and can be run either automatically (input sequences), or by hand (scripts). Each test case represents a "random" walk through the usage model, beginning with the initial state and ending with the termination state. The set of test cases may be as large as the budget and test schedule can tolerate, and bounds on test outcomes can be defined before any test costs are incurred.
• Reference 3 proposes methods for partitioning test cases; allocating testing effort to blocks in the partition so as to minimize reliability estimator variance; and generating test cases from particular blocks in the partition
• The Engineering Practice steps (Reference 2) for Test Planning are:
  1. Random testing should begin only after all non-random testing has been completed
  2. Using the model-derived test case length, estimate and generate the number of random test cases that can be run within existing cost and schedule constraints
  3. Define the "best-case" scenario by assuming that no failures occur during testing and determining the values of product reliability/quality and process efficiency that can be achieved by running the number of generated test cases from step 2
  4. Define the "worst-case" scenario by assuming some failure profile and constructing a failure log based on that profile. Determine the resulting product/process certification measures that can be achieved and compare with actual certification goals to see if the budget and schedule can absorb any disconnects.
  5. Analyze the coverage of model states, arcs and paths that will occur as a result of all testing
  6. If analysis shows that planned testing cannot satisfy model coverage or demonstrated reliability goals, either the goals or the plans must be revised

Table 3.5.10-4: Testing

Characteristics
• The actual behavior of the software under test is compared with the specified behavior (via the usage model) by either manual or automated means. • The behavior of the software is checked on each transition and failures are recorded, as a minimum, by software version, test case number, and transition number. • Once testing has been completed, and in the interests of configuration management and potential software reuse, all test data and test scripts are archived. • It is essential to the statistical integrity of the certification process to maintain experimental control throughout the testing, where control is explicitly defined by adherence to all assumptions regarding the test protocol.

Table 3.5.10-4: Testing (continued)

Characteristics
• The Engineering Practice steps (Reference 2) for Testing are: 1. Each software version should be tested using a unique statistical experiment 2. The usage specification, environmental conditions, and performance evaluation criteria must be held constant for each version of the software that is to be tested 3. Persons performing the test must be properly trained to ensure common understanding of test materials and policies. Human performance should be monitored over the entire test to ensure consistency in test performance. 4. Test cases should be run in the exact order that they are generated, i.e., test cases are not "cherry-picked" for selective application 5. Test results and issues that may affect test judgment should be reviewed and communicated on a regular basis. 6. All failures should be logged 7. For Markov chain testing, maintain at least two testing chains, one for the current software version and one for the history of testing across all software versions. Data from the current-version chain is used for certification and test-stopping decisions and is only valid to estimate the reliability *of that same version*. Data used in **reliability demonstration** models (see Topic 3.6.3) are used to demonstrate the reliability of the software. Data across versions can be used to characterize the testing process. Data used in **reliability growth** models (see Topic 3.6.2.3) are used to measure the effectiveness (or lack thereof) of the process in improving software reliability. 8. If one or more failures occur during testing of the current software version, a decision must be made as to whether the test should be stopped. Stopping decisions may be based on the nature or criticality of the failures, schedules, and organizational policies. Stopping decisions should be guided by the current reliability, statistical confidence, and statistical convergence of the testing chain to the usage chain. 9. If no failures are manifested during testing, stopping decisions should be based on current reliability, statistical confidence, statistical convergence, and remaining budgets and available schedule.

Table 3.5.10-5: Product and Process Measurement

Characteristics
• After completion of testing, results can be used for a number of purposes, including decisions regarding product release, evaluation as to whether the software development process is under statistical control, or assessment of the performance of a new "piece" of technology used in the product • The usage model from which the test cases have been generated is called the "usage chain". A chain that, at least initially, has identical structure is developed to capture and record actual testing experience is called the "testing chain". As testing progresses, the two chains are monitored by tracking measures derived from these two chains. • A reliability measure is calculated from the testing chain, along with statistical confidence intervals. This measured reliability is defined strictly as a function of the failure experience recorded in the testing chain, i.e., no other mathematical assumptions are made. This definition of reliability is applicable whenever one or more failures have been manifested during the test. When testing does not precipitate any failures, statistical distribution models should be used. • Ongoing comparisons between the usage and testing chains are made to quantify the difference (called the discriminant) between expected and actual usage. The trend in the value of the discriminant reveals the rate at which the usage chain and the testing chain are becoming indistinguishable, i.e., the value of the discriminant trends towards zero. • When the value of the discriminant is judged to be suitably low enough to indicate that test performance is sufficiently similar to the modeled field performance, testing should be stopped.

Table 3.5.10-6: Certification

Characteristics
• The certification process involves an ongoing evaluation of the merits of continued testing • Test stopping criteria are based on measures of reliability (probability of taking a random walk through the entire testing chain without failure), statistical confidence, and remaining uncertainty • Decisions to continue testing are based on an assessment that testing goals can still be achieved within the defined schedule and cost constraints • Certification of software is always related to the protocol under which the specific testing as performed, and its related work products. An independent audit of the testing must be possible to confirm the validity of the test results. An independent repetition of the protocol must produce the same conclusions, to within a defined and acceptable statistical confidence, in order for the test results to be considered certified. • When the value of the discriminant is judged to be suitably low enough to indicate that test performance is sufficiently similar to the modeled field performance, testing should be stopped.

Table 3.5.10-7 summarizes the key benefits of statistical usage testing, derived from Reference 2.

Table 3.5.10-7: Summary of Key Benefits of Statistical Usage Testing

Benefit	Comments
Validation of Requirements	• A properly developed usage model represents an easily understood view of the system specification. • Interfaces and requirements are often simplified or clarified when the usage model and its possible inputs, input sequences and outputs are systematically reviewed in the context of operational use
Resource and Schedule Estimation	• Standard calculations that support the usage model provide data that supports resource projections covering effort, cost and schedule, e.g., the minimum number of tests required to cover all usage model states and transitions • Best- and worst-case outcomes from the model based on failures experienced during testing can be bounded through performance of "what-if" analyses
Crafted, Non-Random Test Cases	• Special test cases that may be required by contract or regulations can be performed to ensure that appropriate input sequences are tested • Existing test cases can be mapped to the model to assess omissions or redundancy • The usage model represents the reference model for all required or desired testing
Automated Test Case Generation	• A minimal coverage test script and random test cases (based on the usage probability distribution) can automatically be generated from the usage model • Model coverage testing ensures at least a minimal level of functionality before random testing is initiated • Random testing provides a basis for estimating software operational reliability
Effective, Efficient Testing	• Faults are not equally likely to cause failures, i.e., those on frequently traversed paths have a higher probability of manifesting as failures than those on infrequently traversed paths • Faults are discovered roughly in the same order in which they would cause failures in the field • The test budget is spent in a manner that maximizes the potential increase in operational reliability that results from testing

Table 3.5.10-7: Summary of Key Benefits of Statistical Usage Testing (continued)

Benefit	Comments
Focused Testing (Biased Sampling)	• Usage models support biased sampling of special-interest sequences such as infrequently used, but safety- or mission-critical functions • Separate models can be developed for these functions, or the original model can be suitably modified and the results corrected to remove any bias
Quantitative Test Management	• SUT provides quantitative criteria for management decisions governing completion of testing and system release • Testing sufficiency can be measured as the quantified difference between expected usage (from the model) and tested usage (as recorded)
Estimation of Reliability	• Based on a statistical testing protocol, a valid estimate of expected operational reliability performance can be derived from software testing performance • Actual test results (correct and incorrect performance on each input) are recorded as weights on the usage model, and calculations on the model provide estimated field operational reliability

For More Information:

1. Musa, J.D., “The Operational Profile in Software Reliability Engineering", IEEE Software, Vol. 10, No. 2, March 1993, pp. 14-42
2. Prowell, S.J.; Trammell, C.J.; Linger, R.C.; Poore, J.H.; “Cleanroom Software Engineering: Technology and Process”, Addison-Wesley, 1999, ISBN 0201854805
3. Sayre, K.D., “Improved Techniques for Software Testing Based on Markov Chain Usage Models” (Ph.D. Dissertation), University of Tennessee, Knoxville, December 1999
4. Poore, J. and C. Trammell, ``Engineering Practices for Statistical Testing ," Crosstalk , April 1998

Topic 3.5.11: Operational Profile Testing

Many of the concepts associated with the definition and determination of operational profiles for software were introduced previously in the context of their use to allocate reliability requirements.

As noted in Reference 2, which is the key reference for the bulk of the material presented in this topic, good estimation of software reliability during testing is dependent on accurate knowledge of how a product is going to be used by the customer (or misused by others). Test planning should not start until the operational profile has been defined, since each test should be based on actual operations as implemented within the system, rather than on conceptual functions proposed when the system is initially defined. The operational profile should be considered an integral part of the overall system test plan, and should be used to:

- Allocate efforts during testing
- Select the most appropriate, beneficial test to run
- Determine the order in which the selected tests should be run

Once the operational profile is known, effective tests can be planned to better represent the actual field environment, or appropriate modifications can be made to the reliability estimates (i.e., an appropriate "scaling" or "acceleration" factor can be applied). Reference 2 also suggests that potential savings for a typical project can exceed 50% (or over 10% of a total project cost). Using the operational profile to guide testing can help ensure that, if testing terminates due to schedule constraints, the most-used features of the software will have seen the most comprehensive testing and achieved the maximum reliability level that is practical within the necessary time constraints. During regression testing, the operational profile can be used to allocate a limited number of test cases in accordance with how the customer will use the software, so that those faults that may be introduced and discovered during a change to the program are also those that are most likely to have the greatest impact on the reliability of the software. Alternatively, limits to security-oriented testing can be addressed through allocation of resources to the most severe threats as identified through misuse-case analysis.

The efficiency of operational profile-driven testing is based on the fact that failures are generally identified in order of their frequency of occurrence, so that the faults that cause them can be removed. By exposing the faults and correcting failures associated with the most frequently used operations, the reliability of the software can be rapidly increased.

As discussed in Topic 3.3.1, to develop an operational profile, the steps to be taken include:

- Identification of operations initiators, including (1) users – including misusers -- of the systems, (2) external systems, and (3) operations invoked by the system itself
- Creation of an operations list for each initiator and consolidation of results
- Review of the operations list for correctness and cost realism
- Determination of the occurrence rates of the individual operations values
- Determination of the occurrence probabilities (calculated by dividing the individual occurrence rates by the total occurrence rates of operations values, as appropriate)

Starting from that process, this Topic will address planning for and executing reliability growth testing using operational profiles.

Planning and Preparing for Reliability Growth Testing

Testing based on operational profiles typically involves feature testing (i.e., executing operations individually) followed by load testing (i.e., executing operations simultaneously in realistic field use environment) followed by regression tests (i.e., feature testing after every build involving significant change).

Executing test cases involves executing a number or set of *runs*, where a run is a specific instance of an operation (remember, an operational profile consists of operations along with the associated probability that each operation will be executed when the system is in use). Specifically, a *run* represents an operation, its input variables (both direct and indirect), and values assigned to each input variable.

Planning and preparing for testing involves:

1. Planning/Specifying test cases
2. Specifying test procedures

1. Planning/Specifying Test Cases

If testing a new release of a previously released system, then test cases are being developed for the new operations of the new release. If testing a new system, all operations are new operations, and thus test cases are being developed for all operations. Specifying test cases for the new operations involves the steps outlined in Table 3.5.11-1:

Table 3.5.11-1: Specifying Test Cases for New Operations

Step	Description
Step 1: Planning the number of new test cases for new operations	• The *number of test cases to prepare* is a function of, and ranges between, the *number needed* (based on a test case algorithm) and the number for which *capacity* exists. • The *number needed* is typically estimated as N new test cases per thousand lines of code (KSLOC), based on test performance from previous releases (if reliability of previous releases has been unacceptable, then increase the number). Collard (1) estimates that the range is 2-3/KSLOC for moderate reliability software to 20-33/KSLOC for high reliability software. • The *number* of new test cases *needed* for the current release, N_C, is computed as $N_C = (N / R) \times T$, where o N = total occurrence probability of new operations o R = total occurrence probability of reused operations (N + R = 1.0) o T = total cumulative test cases from all previous releases • *Capacity* is the minimum of (the number of test cases you have time to prepare and the number of test cases you can afford to prepare). Collard (1) estimates that the range is 0.4 to 16 staff hours per test case. • If *number needed* and *capacity* is similar, then *number of test cases to prepare* is the *capacity*. Else, negotiate for additional schedule or budget to develop the *number needed*.
Step 2: Allocating new test cases among base systems and variants of the base system	• The *number of unique new use operation probabilities*, U, for the base system or each variant, is computed as U = F x S, where o F = the fraction of field use for each associated system (base systems or variants) = (each version/variant use in the field/total use in the field). Variations that receive more use in the field will get more new test cases. o S = the sum of occurrence probabilities of new (untested) operations for each associated system (base or variant). Variations that have more usage of untested operations will get more new test cases. • See Table 7.6.10-2 for an example of the 1st release of a system with 2 variants. This table assumes that the Base System will experience 50% of field use (F), Variant 1 will receive 30%, and Variant 2 will receive 20%. Since all operations are new for the Base System, S = 1.0; Variant 1 has 30% of its operations different from the Base Product and Variant 2; Variant 2 has 20% of its operations different from the Base Product and Variant 1. U for each Associated System = F x S. • The *new test case allocation fraction*, L_i, of each associated system i, is computed as $L_i = U_i$ / (sum of all U_i). See Table 7.6.10-2 for an example. • The number of test cases, C_i, of each associated system i, is number of test cases from Step 1 times L_i. See Table 7.6.10-2 with a Step 1 total of 900 test cases.
Step 3: Distributing new test cases among new operations associated with each Associated System	• For each associated system, distribute its number of new test cases to each new operation by multiplying the number allocated to the associated system by the occurrence probability, rounding to the nearest integer. • Suppose Table 3.5.11-2 represented associated systems of the Telephone Billing System from Topic 3.3.1, Table 3.3.1-1. (The Base System might represent the US version, Variant 1 might represent a Canadian Version, and Variant 2 might represent a Mexican version). Table 3.5.11-3 would show the distribution of the Base System test cases (Initial New Test Cases). • Each new operation must be assigned at least 1 test case. See Table 3.5.11-3 (Adjusted New Test Cases). • Identify Critical New Operations (i.e., failure causes a great deal of impact with respect to human life, cost, or system capability). In Table 3.5.11-3, "Recover from system failure" would be such an operation. Increase the number of test cases for such operations proportional to the failure intensity objectives.

Table 3.5.11-1: Specifying Test Cases for New Operations (continued)

Step	Description
Step 4: Detailing new test cases for each new operation	• Selection of values for all direct input variables of each new test case should be achieved by: o Choose among equivalence classes of the operation (a subset of runs of the operation that should yield the same failure behavior because identical processing occurs). o Randomly select levels and values for each direct input variable of the operations, with equal probability among possible choices. o If strong evidence exists of failure-prone values, favor those values o Consider using boundary values for selection. or conditions under which experience has shown to be highly likely to be associated with faults • Prepare test scripts to execute the test cases

Table 3.5.11-2: Example of First Release of a System with Two Variants

Associated System	F	S	U	L	C
Base System	0.5	1.0	0.5	0.735	662
Variant 1	0.3	0.4	0.12	0.177	159
Variant 2	0.2	0.3	0.06	0.088	79
Total, if Applicable	1.0		0.68	1.000	900

Table 3.5.11-3: Distribution of the Base System Test Cases

Operation	Operation Probability	Initial New Test Cases	Adjusted New Test Cases	Critical Operations Adjustment
Residential, no calling plan, paid	0.5940	393	393	393
Residential, national calling plan, paid	0.1580	105	105	105
Business, no calling plan, paid	0.1485	99	99	99
Business, national calling plan, paid	0.0396	26	26	26
Residential, international calling plan, paid	0.0396	26	26	26
Business, international calling plan, paid	0.0099	7	7	7
Residential, no calling plan, delinquent	0.0060	4	4	4
Residential, national calling plan, delinquent	0.0016	1	1	1
Business, no calling plan, delinquent	0.0015	1	1	1
Business, national calling plan, delinquent	0.0006	0	1	1
Residential, international calling plan, delinquent	0.0004	0	1	1
Business, international calling plan, delinquent	0.0003	0	1	1
Recover from hardware failure	0.00001	0	1	3
Total Test Cases		**662**	**665**	**667**

2. Specifying test procedures

A test procedure is a test controller for load test that invokes at various times test cases that are randomly selected from the test case set. Selection from the test case set is based on the test operational profile. Selection of invocation times is based on the total operation occurrence rate and traffic level. Developing the test procedures involves:

- Specify the test operational profile. The last column of Table 3.5.11-3, above, has made test case adjustments to handle minimum test cases and critical operations. Table 3.5.11-4 recalculates the operation probabilities to adjust for these changes – the modified probabilities provide the test operational profile. Note that the sum of the test operational profile probabilities is 1.0. Test procedures will be based on this Test Operation Profile
- Specify the traffic level or average total *operation* occurrence rate as a function of time. In the most general sense, specify a *list of times* from the start of execution of the system and associated new average total operation occurrence rates. One simplification would be to select a constant period (e.g., hourly) or frequency to change the rates, based on what is appropriate for your application. The list of times would

normally cover a fixed duration, such as a day or week. If there are other periodic traffic variations, divide your tests into the same proportions to cover the variations.

Table 3.5.11-4: Recalculation of Operation Probabilities to Adjust for Minimum Test Cases and Critical Operations

Operation	Operation Probability (Original)	Initial New Test Cases	Critical Operations Adjustment	Test Operation Probability
Residential, no calling plan, paid	0.5940	393	393	0.5883
Residential, national calling plan, paid	0.1580	105	105	0.1572
Business, no calling plan, paid	0.1485	99	99	0.1482
Business, national calling plan, paid	0.0396	26	26	0.0389
Residential, international calling plan, paid	0.0396	26	26	0.0389
Business, international calling plan, paid	0.0099	7	7	0.0105
Residential, no calling plan, delinquent	0.0060	4	4	0.0060
Residential, national calling plan, delinquent	0.0016	1	1	0.0015
Business, no calling plan, delinquent	0.0015	1	1	0.0015
Business, national calling plan, delinquent	0.0006	0	1	0.0015
Residential, international calling plan, delinquent	0.0004	0	1	0.0015
Business, international calling plan, delinquent	0.0003	0	1	0.0015
Recover from hardware failure	0.00001	0	3	0.0045
Total Test Cases		**662**	**667**	

- Reproduce, as appropriate, any other significant environmental conditions necessary to make load test represent field use.

Field Experience Use of Operational Profiles

As reported in reference (1), by combining operational profiles with other quality improvement techniques, the user reduced customer reported problems and maintenance costs by 10X, reduced the system test interval by 2X, and improved the product-introduction interval by 30%. The user also reported 10X increase in sales.

Also in reference (1), the author reported that Hewlett Packard, through use of automated testing, failure recording, and Operational Profiles to guide testing; system-test time and system test costs were reduced by more than 50%.

For More Information:

1. Collard, R. 1999. Software Test Estimation, *Proceedings Software Testing Analysis and Review Conference*, May 1999, Orlando, FL. pp. 118-125
2. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
3. Musa, J.D., "The Operational Profile in Software Reliability Engineering: An Overview", Proceedings of the IEEE International Symposium on Software Reliability Engineering, IEEE Computer Society Press, November, 1992, pp. 140-154
4. Musa, J.D., "The Operational Profile in Software Reliability Engineering", IEEE Software, Vol. 10, No. 2, March 1993, pp. 14-42
5. Musa, J.D. *Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition)*. AuthorHouse. 2004. ISBN 1-4184-9388-0
6. Pressman, R.S., "Software Engineering: A Practitioner's Approach", 5th Edition, McGraw-Hill, 1 June 2000, ISBN 0073655783

Topic 3.5.12: Markov Testing

Markov modeling is considered to be an extremely powerful technique for probabilistic modeling and analysis of the random behavior of software over time. It is based on the concept of states and transitions between states. In order to develop a Markov model, the behavior of the system must be broken down into a set of mutually exclusive system states. For software, the states of a system are represented by all unique states that a program may potentially go through based on the activities of the user. The Markov model is uniquely defined by a set of equations that describes, in probabilistic terms, the transitions from one state to another, and an initial probability distribution within each state of the process. For all Markov processes, the transition from the current state to another state depends only on the current state, which embodies the way that the entire past history of the process will affect the future of the process.

> **Discrete-Space Markov Model:** The state space is discrete, either finite or infinitely countable. The Markov process is referred to as a Markov chain. Subcategories are:
>
> > **Continuous-Time Markov Model:** The model allows transitions between states at any time
> > **Discrete-Time Markov Model:** All transitions between states occur at fixed time intervals

Figure 3.5.12-1, slightly modified from Reference 1, provides a simple example of a continuous-time Markov chain representing the operating system reliability for a four-machine PC network. The variable "S_i" represents the number of PCs that are offline due to software failures (e.g., S_0 means that all PCs are operational; S_3 means that 3 of the 4 PCs are "down"). The variable "r_{ij}" represents the transition rates between state S_i and S_j (e.g., r_{03} represents the transition rate between state S_0 and S_3). If unknown, transition rates can be estimated from measured data using the formula:

$$r_{ij} = \frac{\text{Total number of transitions from } S_i \text{ to } S_j}{\text{Cumulative time that the system is in } S_i}$$

It should be noted that, in this example, the set of states and transition rates capture all relevant reliability characteristics of the system at the level at which the system is defined.

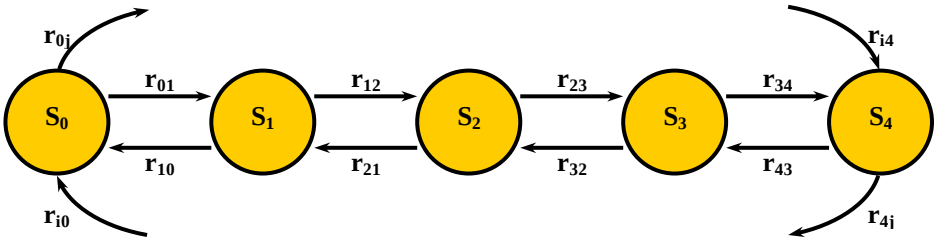


Figure 3.5.12-1: Example Operating System Reliability Markov Chain

For statistical testing of software, all possible uses of the software are represented by the Markov chain model. Each use (or state) of the software will have an associated probability of occurrence. Test cases are derived from a sample population of all possible uses of the software based upon a sample distribution for that state and run against the software under test. The typical reliability metrics of interest include estimated software failure rates and mean time to failure (MTTF). The testing that is performed is evaluated relative to the entire population of software uses (i.e., the entire set of software states) to determine whether testing should continue or be stopped. In the Markov chain usage model, user actions are represented as transitions (or arcs) between states. The probability of a user performing a specific action, given that the software is currently in a specific state, is defined by the transition probabilities within the model. Note that the Markov chain usage model must always contain an "Initiation" state (or "Invoke" state) and a "Termination" state, representing the software state just before the software has been

executed and immediately following software termination, respectively. All test cases must start with the former and end with the latter.

Given a suitable Markov chain usage model, a number of analytically computed results can be developed that are helpful for validating the model, test planning, test monitoring, and evaluating the reliability performance of the software under test. These results include:

- Expected test case length (with associated variance)
- Probability of a state or transition appearing in a test case
- Long-run probability of the software being in a specific usage state

Analysis of the Markov usage chain is valuable for providing insight into how the testing is likely to evolve. This provides testers with the opportunity to proceed with informed test planning and preparation. Table 3.5.12-1, taken from Reference 4, provides some insight into the types of useful results that can be obtained.

Each measure that is indicated in the table is based on an encoded transition matrix, U, with each state representing an index and each transition probability as an entry. The matrix "U" is referred to as a recurrent model, as it causes a new sequence to begin each time the previous sequence ends.

The absorbing model, "U", is a model that represents only single executions of the software, where the Termination state is called absorbing, and all other states in the model are called "transient", where the set of transient states is denoted by the symbol "τ".

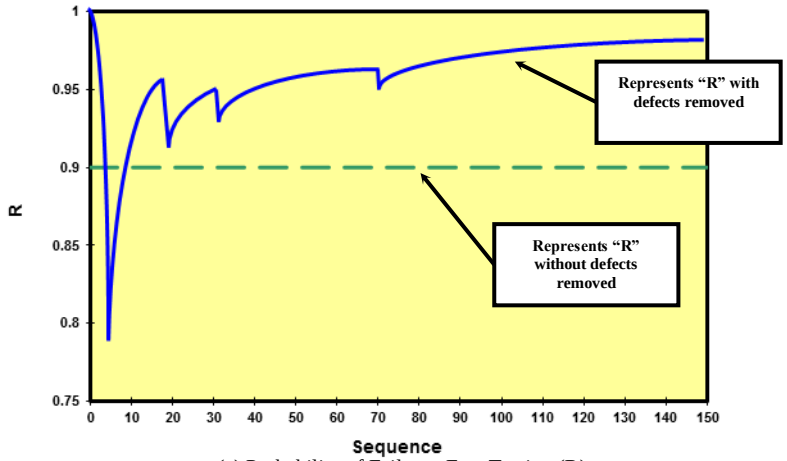
Once the Markov usage chain has been completed, a series of input sequences is stochastically generated and applied to the software under test. This Markov testing chain can be generated either manually or automatically. Which method is used will depend on the nature of the testing environment and the availability of appropriate automated test equipment support. Implied in the approach is the availability of an "oracle" that will be able to (1) make comparisons between the test outputs and the expected results and (2) correctly categorize each test result as a success or a failure.

Table 3.5.12-1: Common Analytical Results for Markov Chains (Reference 4)

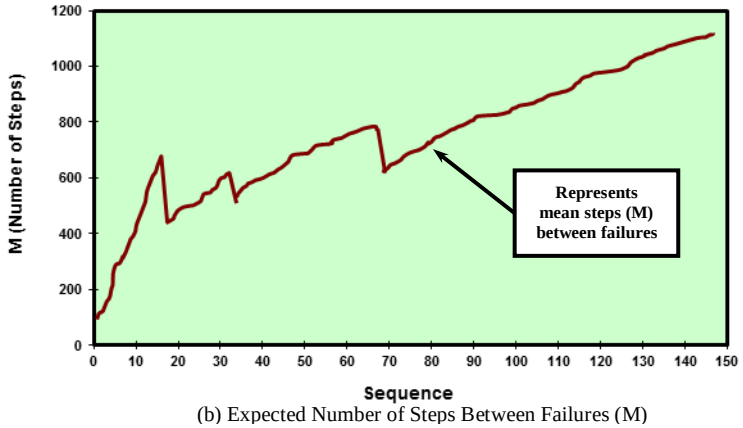
Result	Equation (Probability or Mean)	Interpretation
Recurrent Chain		
Stationary distribution, x	$\boldsymbol{\pi_j = \sum_i \pi_i U_{ij}}$	"π_j" is the asymptotic appearance rate of state "j" in a large number of sequences from transition matrix "U"
Recurrence time for state "j"	$\boldsymbol{m_{jj} = \frac{1}{\pi_j}}$	The mean number of state transitions between occurrences of state "j" in a large number of sequences from transition matrix "U"
No. of occurrences of state "i" between occurrences of state "j"	$\boldsymbol{m_{jj}\pi_i = \frac{\pi_i}{\pi_j}}$	The mean number of occurrences of state "i" between occurrences of state "j"
First passage times	$\boldsymbol{m_{ij} = 1 + \sum_{k=j} U_{ik} m_{kj}}$	The mean number of state transitions until state "j" occurs from state "i"
Absorbing Chain (for initial state "i")		
Single sequence probability for state "j"	$\boldsymbol{y_{ij} = U^a_{ij} + \sum_{k=j} U^a_{ik} y_{kj}}$	The probability that state "j" occurs in a single sequence (i.e., from the initial state to the absorbing state)
No. of sequences to occurrence of state "j"	$\boldsymbol{h_j = \frac{1}{y_{ij}}}$	The mean number of sequences until state "j" occurs
Single sequence probability of arc "jk"	$\boldsymbol{z_{jk} = y_{ij} U_{jk}}$	The probability that arc "jk" occurs in a single sequence (i.e., from the initial state to the absorbing state)
Number of sequences to occurrence of arc "jk"	$\boldsymbol{h_{jk} = \frac{1}{z_{jk}}}$	The mean number of sequences until arc "jk" occurs
No. of occurrences of state "j" in a single sequence	$\boldsymbol{m(j\|i) = \sum_{k\in\tau} U^a_{ik} m(j\|k) + \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}}$	The mean number of occurrences of state "j" in a single sequence

The history of the test at any point in time is a series of input sequences (and usage chain states), and a corresponding sequence of failures, that are specifically identified with the particular sequence and input sequence during which the failure was observed. As failures are discovered and corrected, the software becomes more (or less, depending on the success of the fixes) reliable. Each change to the software (assuming that such changes result in a new software version) has a corresponding test history subset. Reference 4 provides more detailed insight into the construction of the Markov testing chain, including the incorporation of failure data. Where the Markov usage chain represents what would occur in the statistical test if no failures were experienced, the Markov testing chain represents what actually has happened during the test. Initially, the disparity between the two models would be expected to be large, but as testing progresses, the dissimilarity between the models grows smaller (as failures are detected and removed). Ultimately, the results of the Markov testing chain will converge with those of the Markov usage chain, to the point that an analytical approach for determining an appropriate stopping point for test becomes valid. The analytical stopping metric can be based on a reliability measure or on the number of sequences needed/allowed based on available resources (cost, schedule, or personnel).

Reliability measures from Reference 4 that can be derived from Markov testing include probability (R) of failure-free results from the testing chain (similar to overall reliability with fixed-time measurement) and the expected number of steps (M) between failure (analogous to mean time between failure – MTBF – except that steps replace time as the measurement unit). Both metrics are a function of the probabilities assigned to each arc in the usage and testing models, which can be uniformly assigned (as a starting point), weighted by sub-chains, derived from expert opinion or customer surveys, or determined by direct program instrumentation and measurement. A generic graphical representation of each is given in Figure 3.5.12-2.



(a) Probability of Failure –Free Testing (R)



(b) Expected Number of Steps Between Failures (M)

Figure 3.5.12-2: Example Reliability Metrics from Markov Testing

The equations for "R" and "M" are given as follows:

$$\mathbf{R}_{\mathbf{Unin},\mathbf{Term}} = \bar{\mathbf{p}}_{\mathbf{Unin},\mathbf{Term}} + \sum_{\mathbf{j}\in\boldsymbol{\tau}} \bar{\mathbf{p}}_{\mathbf{Unin},\mathbf{j}} \mathbf{R}_{\mathbf{j},\mathbf{Term}}$$

where,

Unin = Uninvoked State (Initiation State, Start State)
Term = Termination State
τ = Set of transient (non-absorbing) states

$$\mathbf{M} = \sum_{\mathbf{i}\in\mathbf{f}_1,\ldots\mathbf{f}_\mathbf{m}} \boldsymbol{\upsilon}_\mathbf{i} \left(\sum_{\mathbf{j}\in\mathbf{u}_1,\ldots\mathbf{u}_\mathbf{n}} \bar{\mathbf{p}}_{\mathbf{ij}} \left(\mathbf{m}_\mathbf{j} + 1 \right) \right)$$

where,

ψ_i = Conditional long-run probability for failure state f_i (given that the process is in a failure state)
m_j = Mean number of steps until first occurrence of any failure state from "j"
$u_1,\ldots,u_n$ is the set of Markov usage chain states
$f_1,\ldots,f_m$ is the set of failure states

The beneficial features of the calculated results from the Markov testing chain are (1) they are based on actual occurrences of failures (no assumptions about failure distributions are required), (2) each generated state is accounted for in the computations (each state sequence contributes according to its length and probability), and (3) each failure is probability-weighted according to its location in the testing chain (failures for high-probability paths will have greater impact on the testing process.

For More Information:

1. Lyu, M.R. (Editor), "Handbook of Software Reliability Engineering", McGraw-Hill, April 1996, ISBN 0070394008
2. Musa, J.D.; Iannino, A.; and Okumoto, K.; "Software Reliability: Measurement, Prediction, Application", McGraw-Hill, May 1987, ISBN 007044093X
3. Sayre, K.D., "Improved Techniques for Software Testing Based on Markov Chain Usage Models" (Ph.D. Dissertation), University of Tennessee, Knoxville, December 1999
4. Whittaker, J.A.; Thomason, M.G., "A Markov Chain Model for Statistical Software Testing", IEEE Transactions on Software Engineering, Vol. 20, No. 10, October 1994, pp. 812-824
5. http://sqrl.eecs.utk.edu/esp/index.html
6. http://www.math.ucdavis.edu/~daddel/linear_algebra_appl/Applications/MarkovChain/MarkovChain_9_18/MarkovChain_9_18.html

Topic 3.5.13: Optimal Release Time

Throughout this Handbook, the subject of decision-making in the context of stopping software testing has been discussed. Topic 3.5.10 addressed the issue in the context of a statistically-based discriminant that, as it approaches zero, corresponds to suitable correlation between the usage model and the testing results to warrant stopping the test. Topic 3.5.12 provided quantifiable measures for probability of failure-free testing (R) and the mean number of steps between failure (M), where a decision to stop testing can be based on achieving a specific level of reliability or a predetermined mean number of steps between failure. In general, any decision to stop software testing is ultimately based on a business decision that is contractual, schedule-related, cost-related, or performance-related.

This topic presents a concept that can be used to determine the optimal release time for software based on cost. It is not necessarily limited to decisions regarding test length, but is appropriate for that purpose.

Consider a software system whose failures are modeled by a Nonhomogeneous Poisson Process with a failure intensity function $\lambda(t)$. The question to be addressed here is when should system testing be terminated and the software released. This decision is based on minimizing the sum of system testing and operations cost.

Let: c_1 be the cost of removing a defect during testing
c_2 be the cost of removing a defect after release
c_3 be the cost of testing per unit time.

A reasonable assumption is that the failure intensity function is decreasing (reasonable if testing is, indeed, identifying and removing defects without introducing new defects at a faster rate than they are removed). This Optimal Release Time model is valid if and only if the cost of removing a defect after release is greater than the cost of removing a defect during testing ($c_2 > c_1$).

Table 3.5.13-1 provides appropriate release criteria. In practice, estimates of the parameters of the failure intensity function will be updated continuously. An estimate of the optimal release time should be updated accordingly.

Table 3.5.13-1: Release Criteria

Condition	Decision
$(c_2 - c_1)\lambda(t) > c_3$	Continue system test
$(c_2 - c_1)\lambda(t) \le c_3$	Release software

As an example, consider the Musa-Okumoto logarithmic Poisson model. The failure intensity function for this model is given as:

$$\lambda(t) = \frac{\lambda_0}{\theta \lambda_0 t + 1}$$

where,

$\lambda(t)$	=	failure intensity at time, t, in failures per CPU-hour
λ_0	=	initial failure intensity at the start of execution (f/CPU-hour)
θ	=	failure intensity decay parameter
t	=	time, t, at which the failure intensity is to be calculated (CPU-hours)

Solving this equation for t and factoring in the relevant cost factors yields the following solution for the optimal release time, t^*:

$$t^* = \frac{1}{\theta \lambda_0}\left(\frac{c_2 - c_1}{c_3}\lambda_0 - 1\right)$$

As an example, consider the situation where the initial failure intensity (λ_0) is 20 failures per CPU hour and the failure intensity decay parameter (θ) is 0.04 per failure. Additionally, assume (hypothetically) that the cost of removing a defect after release (c_2) is \$10,000, the cost of removing a defect during testing (c_1) is \$1,000, and the cost of testing per CPU-hour (c_3) is \$200. The optimal release time is calculated as:

$$\mathbf{t}^* = \frac{1}{(0.04)(20)}\left(\left(\frac{10000 - 1000}{200}\right)(20) - 1\right)$$

$$\mathbf{t}^* = \frac{1}{0.80}\big((45)(20) - 1\big) = (1.25)(899) = 1124\,\text{CPU - Hours}$$

Suppose that improvements in defect detection/elimination improved the value of θ to 0.10 per failure. The resulting optimal release time would be:

$$\mathbf{t}^* = \frac{1}{2}\big((45)(20) - 1\big) = (0.50)(899) = 450\,\text{CPU - Hours}$$

If the initial failure intensity could be improved through robust software design to 4 failures per CPU-hour, the resulting optimal release times for the two cases illustrated above would become 106 CPU-hours and 42.5 CPU-hours, respectively.

For More Information:

1. Vienneau, R.L., "The Cost of Testing Software," Proceedings of the Annual Reliability And Maintainability Symposium, Orlando, Florida, 29-31 January 1991

Topic 3.5.14: Security Testing

Security testing, in contrast to other types of testing treated in this Handbook, is more about what the software *ought not* to do rather than what it *ought* to do.

Security issues arise from that fraction of software defects that are termed *vulnerabilities*, which could be exploited to adversely affect confidentiality, integrity, or accessibility of a system or its data.

Rather than being driven by customer- or user-supplied requirements, security testing is typically mapped against anticipated attacks on the system. Hence, the development of *misuse* (or *abuse*) *cases* to describe conditions under which attackers might threaten the system, in contrast to the traditional use cases, which describe "normal" interaction patterns.

Just as traditional usability and reliability testing need a proper context for their design and interpretation, so too security testing needs its context if it is to provide useful insights. Threat modeling (Reference 1) or security testing can be considered analogous to the development of operational profiles in reliability testing.

Identifying potential threats to security is inherently more complex and uncertain than working within a well-defined community of stakeholders, all of whom wish the system to work successfully. First, the value of the system – its appeal to attackers – must be characterized across a range of potential misusers. Further, different attackers will themselves have different definitions of success, such as the extent to which they wish to remain undetected or anonymous.

Any testing must always be considered as but one tool in support of risk management. Recent guidance from the National Institute for Standards and Technology (Reference 2), for instance, places security testing within the larger context of *security control assessment*, defined as "the testing and/or evaluation of the management, operational, and technical security controls to determine the extent to which the controls are implemented correctly, operating as intended, and producing the desired outcome with respect to meeting the security requirements for an information system or organization."

Unfortunately, security requirements are not typically specified and security expectations are not usually explicitly aligned with organizational needs and strategies, as Figure 3.5.14-1 from Reference 2 idealizes.

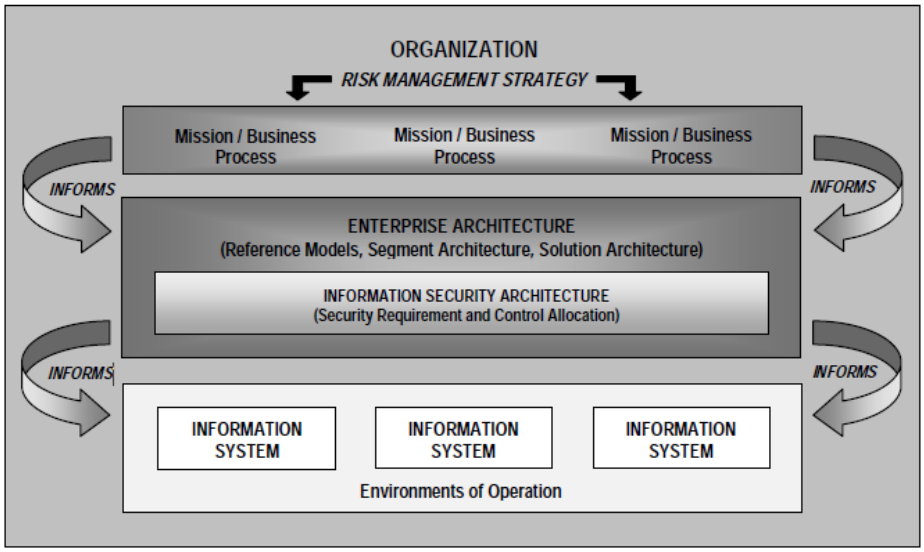


Figure 3.5.14-1: Information Security Requirements Integration (Reference 2)

Instead, the challenge all too often is simply to understand where the organization is most vulnerable and then to design and conduct security testing accordingly. As with any other testing, the limitations of schedule and budget force prioritization in order to maximize the value of the testing.

Another NIST publication (Reference 3) emphasizes various limitations of the testing approach and indicates that testing is best combined with a wider array of assessment techniques. For instance, "testing is less likely than examinations to identify weaknesses related to security policy and configuration."

Reference 3 characterizes penetration testing as support of *target vulnerability validation* and brackets its application with "password cracking" and "social engineering". Indeed, the greatest weakness in any security configuration is likely to be the human element, and testing should assess exposures due to attackers' acquiring insider knowledge or impersonating legitimate users.

To the extent that security measures primarily consist of building defensive barriers, then the prerequisite for any security assessment would *be penetration testing.* This type of testing is meant to determine the capabilities required to breach those barriers.

A helpful historical survey (Reference 4) asserts that "penetration testing has indeed advanced significantly but is still not as useful in the software development process as it ought to be" due to a number of limitations. Most penetration testing is typically done far too late in the software development life cycle and without sufficient sensitivity to the wider range of business risks. Traditional organizational responsibilities and reporting chains also mean that test result analyses "generally prescribe remediation at the firewall, network, and operating system configuration level … [which are not] truly useful for software development purposes."

Reference 4 provides helpful discussions of specific vulnerability scanning and testing tools (host- or network-based) and techniques. It advocates penetration testing that is less "black box" (ignorant of system internals) and more "clear box" (sometimes called "white box") in which design and implementation details are visible to the tester, allowing exploration of more potential weak spots.

Note that "white box" testing is not to be confused with "white hat" testing. Security testing is often differentiated as being conducted as either overt ("white hat") or covert ("black hat").

A more complex representation (and less standard terminology) of the possible permutations of mutual knowledge is shown in Figure 3.5.14-2 from Reference 5

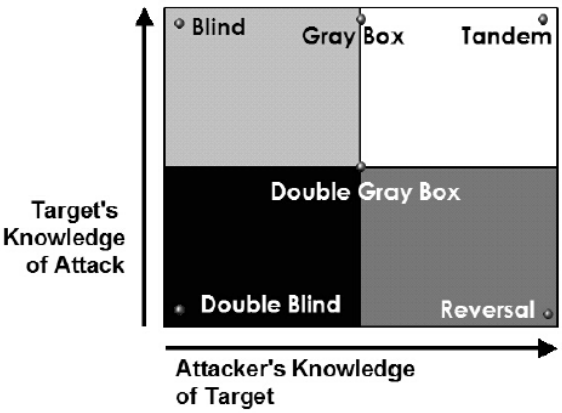


Figure 3.5.14-2: Target-Attacker Matrix

See [6] for further details on design-based security testing, which may involve analyzing data flow, control flow, information flow, coding practices, and exception and error handling within the system. It recommends test-related activities across the development life cycle:

- *Initiation Phase* should include preliminary risk analysis, incorporating history of previous attacks on similar systems.
- *Requirements Phase* involves establishing test management processes and conducting more detailed risk analysis.
- *Design Phase* allows focus of test resources on specific modules, such as those designed to provide risk mitigation.
- *Coding Phase* permits functional testing to begin at the unit level as individual modules are implemented.
- *Testing Phase* moves from unit testing through integration testing to complete system testing.
- *Operational Phase* may begin with beta testing and continues to require attention as deployment may involve configuration errors or encounters with unexpected aspects of the operational environment.

See Reference 7 for an extensive discussion of software assurance tools and techniques, with links to numerous other resources and to a reference dataset of security flaws and associated identification test cases.

No measures of *security test coverage* seem particularly helpful. One could test against a "top twenty-five" list of known vulnerabilities (Reference 8)**,** but no Pareto-like analysis has ever been published to indicate if exploiting the "top twenty-five" results in 80% of security breaches … or 8% … or any other proportion.

Similarly, one might wish to engage in *security growth modeling*, analogous to reliability growth modeling [9], but little empirical data and no significant supporting analyses have been published.

Threat modeling explores a range of possible attackers, all with different capabilities and incentives. These characteristics might be helpfully profiled in terms of knowledge, skills, resources, and motivation:

- What is the distribution of *knowledge* about existing vulnerabilities?
- How likely is an attacker to possess the *skills* required to exploit a given vulnerability?
- How extensive are the *resources* (access, computing power, etc.) that might be brought to bear?
- What *motivations* would keep a given attacker on task to successful completion of the attack?

Results of test cases need to be considered with more nuance than simply noting the success or failure of breaching security. They must be calibrated in terms of these same aspects:

- What *knowledge* about a given vulnerability was assumed in the test case?
- What specific *skills* and skill levels were employed within the test?
- How extensive were the *resources* that were required to execute the test?
- What *motivations* of an attacker would be sufficient to persist and produce a similar result?

Risk-based security testing (Reference 10) is concerned not simply with the probability of a breach but, more importantly, with the nature of any breach. What are the goals of different attackers and what might be the consequence of their actions?

- A range of security tests might, for instance, indicate the probability of a successful denial-of-service attack as 20%, of a confidentiality violation as 10%, and of an undetected data integrity manipulation as 5%.

- If Attacker A were interested in extorting protection payment to forego a $1,000,000 opportunity cost due to website unavailability, then the business would be facing a risk exposure of 20% of that potential loss -- $200,000.

- Attacker B, intent on revealing confidential information that would cost $5,000,000 in regulatory fines and legal expenses, would represent a risk exposure of 10% of that value -- $500,000.

- Finally, if a competitive advantage of $20,000,000 might be gained by Attacker C successfully corrupting business-sensitive data, then that risk exposure would be greatest even with 5% as the lowest probability of occurrence -- $1,000.000.

The next step would be to analyze the *return on security investment* and allocate resources accordingly. Reducing risk exposure might be accomplished by any combination of reducing the probability of the occurrence (*risk avoidance*) and reducing the consequences should the event occur (*risk mitigation*).

Consider defending against Attacker A. Perhaps $25,000 is budgeted for security improvement. If that amount was invested in risk avoidance, say by strengthening website defenses, it might reduce the probability of a successful denial-of-service attack from 20% to 15%, representing a risk exposure reduction of 5% of the potential $1,000,000 loss and yielding a return on investment of $50,000/$25,000 = a ratio of 2.

An alternative investment in risk mitigation, say by decreasing incident recovery time, might lower the cost of a successful attack by $400,000. That return on investment would be calculated for a risk exposure reduction of $80,000 (given the unchanged 20% probability of occurrence) divided by the $25,000 allocated – a more attractive return ratio of 3.2.

Of course, an even better return might be found by some optimal combination of investments in both risk avoidance and risk mitigation.

Reference 10 concludes that "although it is strongly recommended that an organization not rely exclusively on security test activities to build security into a system, security testing, when coupled with other security activities performed throughout the SDLC, can be very effective in validating design assumptions, discovering vulnerabilities associated with the application environment, and identifying implementation issues that may lead to security vulnerabilities."

For More Information:

1, Frank Swiderski and Window Snyder. *Threat Modeling*. Redmond, Washington: Microsoft Press. 2004.

2. NIST Special Publication 800-39, *Managing Information Security Risk: Organization, Mission, and Information System View* (March 2011).

3. NIST Special Publication 800-115, *Technical Guide to Information Security Testing and Assessment* (September 2008).

4. Kenneth R. van Wyk. Adapting Penetration Testing for Software Development Purposes. (https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/penetration/655-BSI.html).

5. Institute for Security and Open Methodologies. *OSSTMM 3 – The Open Source Security Testing Methodology Manual* (http://www.isecom.org/mirror/OSSTMM.3.pdf)

6. Girish Janardhanudu. White Box Testing (https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html).

7. Software Assurance Metrics And Tool Evaluation (http://samate.nist.gov/)

8. *2011 CWE/SANS Top 25 Most Dangerous Software Errors* (http://cwe.mitre.org/top25).

9. M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*: McGraw-Hill and IEEE Computer Society Press, 1996.

10. C. C. Michael and Will Radosevich. Risk-Based and Functional Security Testing. (https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing/255-BSI.html)

11. Julia H. Allen, Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software Security Engineering: A Guide for Project Managers*. Upper Saddle River, N.J.: Addison-Wesley. 2008.

12. Chris Wysopal, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. *The Art of Software Security Testing: Identifying Software Security Flaws.* Upper Saddle River, N.J.: Addison-Wesley. 2007.

Topic 3.6: Reliability Growth and Reliability Demonstration/Qualification Testing

Topic 3.6.1: Overview

Before we can understand some of the conditions under which to apply, consider, or not apply reliability growth testing (RGT) and reliability demonstration/qualification testing (RDT/RQT, which includes Production Reliability Acceptance Tests, or PRAT), it is first necessary to understand the basic philosophical differences between the two test types. These basic differences are summarized in Table 3.6.1-1, and discussed below.

Table 3.6.1-1: Differences Between RGT and RDT/RQT

Characteristic	RGT	RDT/RQT
Philosophy	Use test/failure data to grow reliability	Use test/failure data for accept/reject decisions
Length	Minimum length is instantaneous MTBF corresponding to "MTBF-Specified"	Statistically-based test plan per MIL-HDBK-781
Environment	Combined temperature/vibration (simulated field) or representative operating profile	Combined temperature/vibration (simulated field) or representative operating profile
Failures	Failures are good	Failures are bad
Representative Test	Minimum 385 cumulative operating hours at 0.40 growth rate with $\leq$ 1 failure	MIL-HDBK-781, Test Plan IV-D test length can range from 1800 hours (no failures) to 6240 hours (for 5 or more failures) before accept decision is reached.

The philosophy of RGT is to use the generated test and failure data to identify failure modes/mechanisms and find/fix design root failure causes (i.e., failure mode mitigation), thereby improving the inherent reliability. In this context, failures are good and they should be encouraged.

The philosophy of RDT/RQT is to use test and failure data to reach statistically valid decisions regarding whether an item has achieved its specified reliability or not (that is, it is either accepted or rejected). In this context, failures are bad if they ultimately result in a reject decision.

RGT can be considerably shorter than the statistically-based test plans upon which RDT/RQT are based. This is because, for RGT, the test length is determined by the minimum amount of time required to grow the instantaneous MTBF to its corresponding requirement. For an aggressive reliability growth program which can achieve a growth rate of 0.40, a 385-hour test could conceivably be run to achieve an instantaneous MTBF of 640 hours. A statistical test plan per MIL-HDBK-781, upon which RDT/RQT is based, typically requires multiples of the specified MTBF as its test time. The size of this multiplier depends on the degree of Producer's or Consumer's risk assumed, the amount of statistical confidence you want to have in the test results and, for sequential tests, the number of failures that occur upon which an accept or reject decision is to be based. An item having a specified MTBF of 640 hours, then, may range from a minimum of 1800 test hours to over 6200 test hours before an accept or reject decision is reached. Test length is not the most important distinction between the two types of tests, however.

Both RGT and RDT/RQT need to be performed using an environment that simulates the environmental and operational stresses or the operational profile that will be experienced under normal use conditions. The differences between RGT and RDT/RQT help to dictate some situations where either one or both should be applied, considered or not applied. Table 3.6.1-2 highlights some of these scenarios.

Table 3.6.1-2: RGT and RDT/RQT as a Function of Test Item Characteristics

Test Item Characteristics	Reliability Growth Test			Reliability Demonstration/ Qualification Test (including PRAT)		
	Apply	Consider	Don't Apply	Apply	Consider	Don't Apply
Challenge to SOA	X			X		
Use environment or operating profile is "severe"	X			X		
One-of-a-kind item			X		X	
High production quantities	X			X		
Use environment is benign			X		X	
Critical performance requirements	X			X		
Design flexibility exists	X			X		
Design flexibility does not exist			X		X	
Schedule limitations			X		X	
Funding limitations			X		X	
Required reliability measure of item is very high			X			X

RGT and RDT/RQT should both be applied (1) when there is a challenge to the technological state-of-the-art, (2) when the use environment is relatively severe, (3) when there are high production quantities, (4) where critical performance requirements must be met, and (5) whenever design flexibility exists.

Challenges to the state of the art implies that failure modes and mechanisms are unknown, and RGT is useful in that context, as well as providing increased confidence that specified MTBF requirements can be met. The statistical confidence that the specified MTBF requirement is, in fact, met comes from RDT/RQT. This same "feeling" of confidence applies to the other scenarios where both tests are appropriate.

For one-of-a-kind items, benign use environments, inflexible designs, and limited schedules or funding, RGT should not be applied, but RDT/RQT should be considered. If design flexibility does not exist, then whatever is found out from RGT is moot. Also, given that failures during RGT may require an extended shutdown of the reliability growth process while root failure causes are found, designs modified and corrective actions verified, limited or restrictive schedules and funding are not an appropriate scenario for performing RGT. The use of RDT/RQT should be considered in these scenarios, however, particularly if there is a valid need for empirically demonstrating that the reliability requirement has been met, i.e., when analytical proof of meeting the requirement may not be sufficient to satisfy the customer.

Finally, in scenarios where the required reliability measure (MTBF, R(t), Availability, etc.) is very high, neither RGT or RDT/RQT should be applied, as the significant investment in time, money and resources required to run these tests is not justified.

Topic 3.6.2: Reliability Growth Testing

Reliability growth is the intentional positive improvement (negative reliability "growth" can also occur) that is made in the reliability of a product or system as defects are identified, detected, analyzed for root cause, and mitigated. The process of defect removal can be (1) ad hoc, as they are when discovered during design and development, (2) a function of an informal test-analyze-and-fix process (and all of its possible permutations), or (3) as a result of formalized reliability growth testing (RGT). The rate at which inherent design reliability grows is a function of how fast defects or failures can be detected, their root cause(s) removed, and their corrective actions identified, implemented and verified.

Formal RGT is performed to evaluate current reliability, identify and eliminate hardware and software faults, and forecast future product or system reliability. Reliability metrics are compared to planned periodic intermediate goals to assess progress. Depending on the progress (or lack thereof) achieved, resources can be allocated (or re-allocated) to meet those goals in a timely and cost-effective manner. RGT should always be performed under similar end-use conditions (environmental stresses, operating profiles, etc.) as those expected in the field in order to be effective.

Who pays for the RGT? Does the customer end up paying more? The usual case is that the customer pays for the RGT as an additional reliability program cost and in stretching out the schedule. The savings in maintenance costs should exceed the additional initial investment, resulting in a net savings in system total life cycle cost (LCC). The amount of these savings is dependent on the quantity of systems to be fielded, the maintenance concept, the sensitivity of LCC to reliability and the level of development required.

Does RGT allow manufacturers to "get away" with a sloppy initial design because they can fix it later at the customer's expense? It has been shown that unforeseen problems account for 75% of the failures due to the complexity of today's products. Too low an initial reliability (resulting from an inadequate manufacturer design process) will necessitate an unrealistic growth rate in order to attain an acceptable level of reliability in the allocated amount of test time. The reliability growth test should be considered as an organized search and correction process for reliability problems that allows problems to be fixed when it is least expensive. It is oriented towards the efficient determination and verification of corrective action. Solutions are emphasized.

Should all system development programs have some sort of growth test? The answer to this question is "yes" in that all development programs should analyze and correct failure modes when they are identified both prior to and during testing. A distinction should be in the level of formality of the test program. The less challenge there is to the state-of the-art, the less formal (or rigorous) a reliability growth test needs to be. In the case of off-the-shelf items (NDI/COTS/GOTS/OSS) that are integrated into a system, design flexibility to correct reliability problems is constrained to newly developed interfaces between the "boxes" making up the system. A rigorous growth program would be inappropriate for these items, but a failure reporting, analysis and corrective action system (FRACAS) should still be implemented, with cooperation from suppliers on determination and correction of root failure causes for their products.

What reliability growth model(s) should be used? The model to be used is the simplest one that does the job. Two of the most widely used models, the Duane Model and the AMSAA Model (developed by Dr. Larry H. Crow, formerly of the Army Materiel Systems Analysis Activity), each have advantages; the Duane being simple with parameters having an easily recognizable physical interpretation, and the AMSAA having rigorous statistical procedures associated with it. MIL-HDBK-189 suggests the Duane for planning and the AMSAA for assessment and tracking. When an RQT is performed, the RGT should be planned and tracked using the Duane Model; otherwise, the AMSAA Model is recommended for tracking because it allows for the calculation of confidence limits around the data. AMSAA has also developed reliability growth projection models including PM2, typically used for planning, and the AMSAA Maturity Projection Model (AMPM), typically used to project reliability growth based upon testing completed to date. Development of specialized reliability growth models for software products is also an area of ongoing research.

Topic 3.6.2.1: Duane and Crow/AMSAA Models

For high-level RGT, where hardware and software elements are integrated (e.g., subsystem testing or system integration), the Duane or Crow/AMSAA models are typically used to assess reliability growth. The Duane model is simple, having parameters that allow for an easily recognizable physical interpretation. The Crow/AMSAA model has more rigorous statistical procedures associated with it that allow the calculation of confidence intervals around the growth curve as testing progresses. Both models are presented here in the context of failure rate, although it is a more common practice for products and systems to use the models based upon mean time between failure (MTBF). It should be noted that Dr. Crow has revised his model on several occasions, publishing his "Extended Reliability Growth Model" in 2004 (Reference 1) and his "Extended Continuous Evaluation Reliability Growth Model" in 2010 (Reference 2). AMSAA has also released its own reliability growth methodology (Reference 3), which has been implemented into their AMSAA Projection Methodology (PM2) and AMSAA Maturity Projection Model (AMPM) software tools. The AMPM model is discussed in Topic 3.6.2.2.

Duane Model

Table 3.6.2.1-1 summarizes the options, required inputs and calculated outputs associate with the use of the basic Duane reliability growth model.

Table 3.6.2.1-1: Duane Reliability Growth Planning Option, Required Inputs and Calculated Outputs

Model	Options	Required Inputs	Calculated Outputs
Duane Postulate (MIL-HDBK-189 Reliability Growth Planning Model)	• Construct idealized system reliability growth curve, identify test time and growth rate required to improve system reliability • **Option 1**: Develops a set of reliability growth planning steps where the expected MTBF for the first and last test phases are required inputs • **Option 2**: Develops a set of reliability growth planning steps where the expected MTBF for the first test phase and the expected final MTBF on the planning curve are required inputs	• Initial failure rate or MTBF • Assumed reliability growth rate • The planned number of test phases • The cumulative expected amount of test time at the end of each test phase • The expected average MTBF for the first test phase • The expected average MTBF for the last test phase (Option 1 only) • The expected final MTBF value on the planned growth curve (Option 2 only)	• Number of cumulative test hours to achieve the required reliability • Total expected number of failures • The expected reliability growth rate • Model scale parameter • The expected average MTBF for each test phase • A plot of the average MTBFs together with the smooth idealized curve • A plot of the average Failure Rates and the smooth idealized curve

Table 3.6.2.1-2 provides a brief overview of the elements of the Duane model.

Table 3.6.2.1-2: Characteristics of the Duane Reliability Growth Model

Parameter	Comments
Growth Rate (α)	**For failure rate, the growth rate is**: $\alpha = -m = -\dfrac{\sum X_i Y_i - \left(\sum X_i \sum Y_i\right)/n}{\sum X_i^2 - \left(\sum X_i\right)^2/n}$ where, Y_i = log of cumulative failure rate at time "T" and failure "i" X_i = log of the cumulative time at failure "i" n = total number of failures **For MTBF, the growth rate is**: $\alpha = \dfrac{\Delta \text{MTBF}}{\Delta \text{time}}$ Based on experience within the industry, growth rates in the Duane model never exceed 0.60, and growth rates above 0.50 are rare. A growth rate of 0.25 to 0.4 is average for most projects. Also, growth rates can be determined directly from the plotted test data (log failure rate vs. log cumulative time)
Cumulative Failure Rate (λ_{cum})	The cumulative failure rate represents the measured failure rate, over time, as failures occur. It is calculated using the formula: $\lambda_{cum} = KT^{-\alpha}$ where, K = a constant which is a function of the initial failure rate T = test time α = growth rate The value used for K is calculated as: $K = 10^{\frac{\sum Y_i + \alpha \sum X_i}{n}}$
Cumulative MTBF ($MTBF_{cum}$)	The formula for cumulative MTBF is simply the inverse of the formula for the cumulative failure rate, i.e.: $MTBF_{cum} = \dfrac{1}{K} T^{\alpha}$
Instantaneous Failure Rate (λ_{inst})	The instantaneous failure rate (and MTBF, for that matter) is the mathematical representation of the failure rate (or MTBF) if all previous failure occurrences are corrected. Its formula is: $\lambda_{inst} = K(1-\alpha)T^{-\alpha}$ or $(1-\alpha)\lambda_{cum}$
Instantaneous MTBF ($MTBF_{inst}$)	The formula for instantaneous MTBF is, again, the inverse of the formula for the instantaneous failure rate, i.e.: $MTBF_{inst} = \dfrac{\frac{1}{K}T^{\alpha}}{1-\alpha}$ or $\dfrac{MTBF_{cum}}{1-\alpha}$

Idealized Growth Curve

An Idealized Growth Curve is a planned growth curve that consists of a single smooth curve based on initial conditions, an assumed growth rate, and/or planned management strategy. Any or all of these may be subjective and, therefore, will have a significant impact on the relationship between the planned growth curve and the achieved reliability growth that is tracked against it, depending on how "good" the inputs are into the planning curve process. The Idealized Growth Curve is a strict mathematical function of the input parameters across the measure of test duration (e.g., time, distance, trials), thus the name - Idealized. **No program can be expected to assume this exact mathematical ideal shape, but it is useful in setting interim goals.**

Using the Duane reliability growth model the idealized curve is given by:

$$M(t) = \begin{cases} M_I & , 0 < t < t_I \\ \left(\dfrac{M_I}{1-\alpha}\right)\left(\dfrac{t}{t_I}\right)^{\alpha} & , t > t_I \end{cases}$$

where M(t) is the MTBF at time t, M_I is the average MTBF over the initial test phase, t_I is the duration of the initial test phase, and α is the expected growth rate.

Example 1: How to Determine the Idealized Growth Curve

Suppose that the initial MTBF for the system is estimated to be 45 hours and a final MTBF of 110 hours is desired after 10,000 hours of testing. For this program, the first test phase is 1,000 hours. This is the point where delayed fixes will first be introduced into the system. Further, some reliability growth is planned during the first test phase so that an average MTBF = 50 hours is anticipated during the first phase. Determine the idealized growth curve.

Solution

In order to determine the idealized growth curve, the growth rate parameter, α, must be determined. Rearranging the equation above yields the following:

$$\frac{M(t)}{M_I} = \left(\frac{1}{1-\alpha}\right)\left(\frac{t}{t_I}\right)^{\alpha}$$

Taking the natural logarithms

$$\ln\left(\frac{M(t)}{M_I}\right) = \alpha \ln\left(\frac{t}{t_I}\right) - \ln(1-\alpha)$$

The growth rate is determined by substituting for the known parameters in the above equation and solving for α. Note, this solution may have to be performed iteratively.

Using the information provided in Example 1, the growth rate parameter is determined to be $\alpha = 0.23$, which is found as the solution to:

$$\ln (110 /50) = \alpha \ln (10{,}000/1000) - \ln (1 - \alpha)$$

Therefore, if a growth rate of $\alpha = 0.23$ is considered acceptable, the idealized growth curve is given by:

$$M(t) = \begin{cases} 50 & , 0 < t < 1000 \\ \left(\frac{50}{1-0.23}\right)\left(\frac{t}{1000}\right)^{0.23} & , t > 1000 \end{cases}$$

A plot of the idealized growth curve is shown in Figure 3.6.2.1-1.

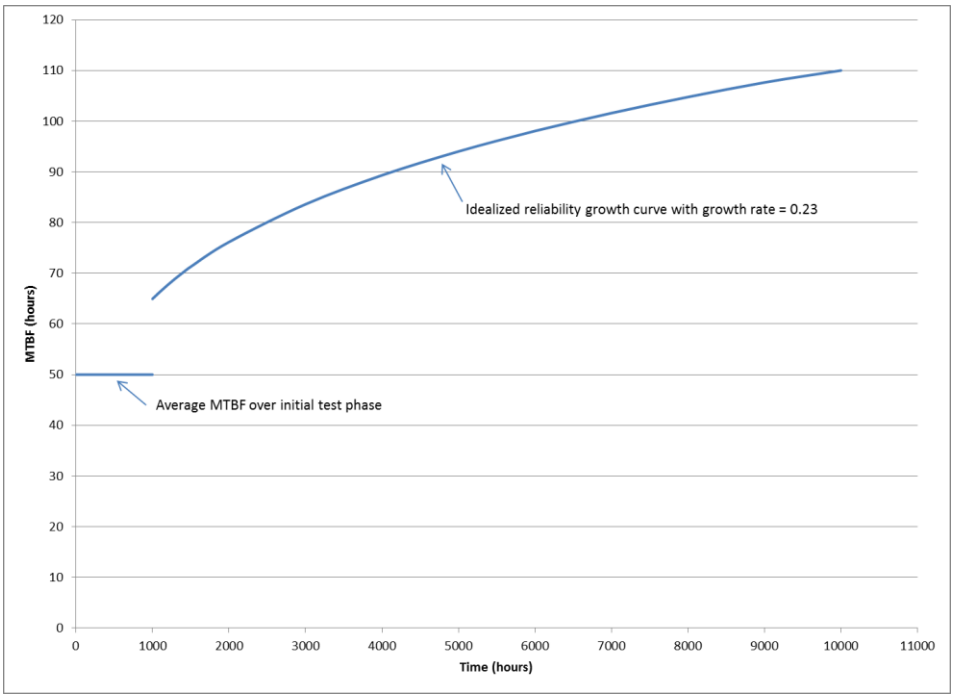


Figure 3.6.2.1-1. Example 1 Idealized Growth Curve

Example 2: How to Determine the MTBF for a Test Phase

Continuing Example 1 above, the first test phase was defined from 0 to 1,000 hours. Suppose that the program consists of four additional test phases at 1000-2500 hours; 2500-5000 hours; 5000-7000 hours; and 7000-10,000 hours. To determine the average MTBF's to be expected over these periods if reliability growth follows the idealized curve use:

$$M(t) = \begin{cases} 50 & , 0 < t < 1000 \\ \left(\frac{50}{1-0.23}\right)\left(\frac{t}{1000}\right)^{0.23} & , t > 1000 \end{cases}$$

It can be shown that the number of failures expected to be observed by time t is given by:

$$N(t) = \lambda_I \alpha_I \left(\frac{t}{t_I}\right)^{(1-\alpha)}$$

where λ_I is the average failure rate over the first test phase, and t_I is the duration of the first test phase. The average number of failures occurring during the i^{th} test phase is then given by:

$$H_i = N(t_i) - N(t_{i-1})$$

The average MTBF over the i^{th} test phase is then given by:

$$MTBF_i = \frac{D_i}{H_i}$$

where D_i is the duration of the i^{th} test phase.

Using the initial failure rate of 0.02 failures per hour, duration of the initial test phase of 1000 hours and a growth rate of 0.23, the average number of failures and the average MTBF for each of the defined test phases of this example can be calculated. The results of these calculations are presented in Table 3.6.2.1-3.

Table 3.6.2.1-3: Average Number of Failures and MTBF for Each Test Phase of Example 2

Phase i	Test Phase Start-End (Hours)	t_i (Hours)	Test Phase Duration D_i	Number of Failures at t_i N_i	Average Number of Failures in Test Phase H_i	Average MTBF Over Test Phase (Hours)
1	0-1000	1000	1000	20	20	50
2	1000-2500	2500	1500	40.5	20.5	73
3	2500-5000	5000	2500	69.1	28.6	87
4	5000-7000	7000	2000	89.5	20.4	98
5	7000-10000	10000	3000	117.8	28.3	106

A plot showing the idealized curve and the average test phase MTBF is provided in Figure 3.6.2.1-2.

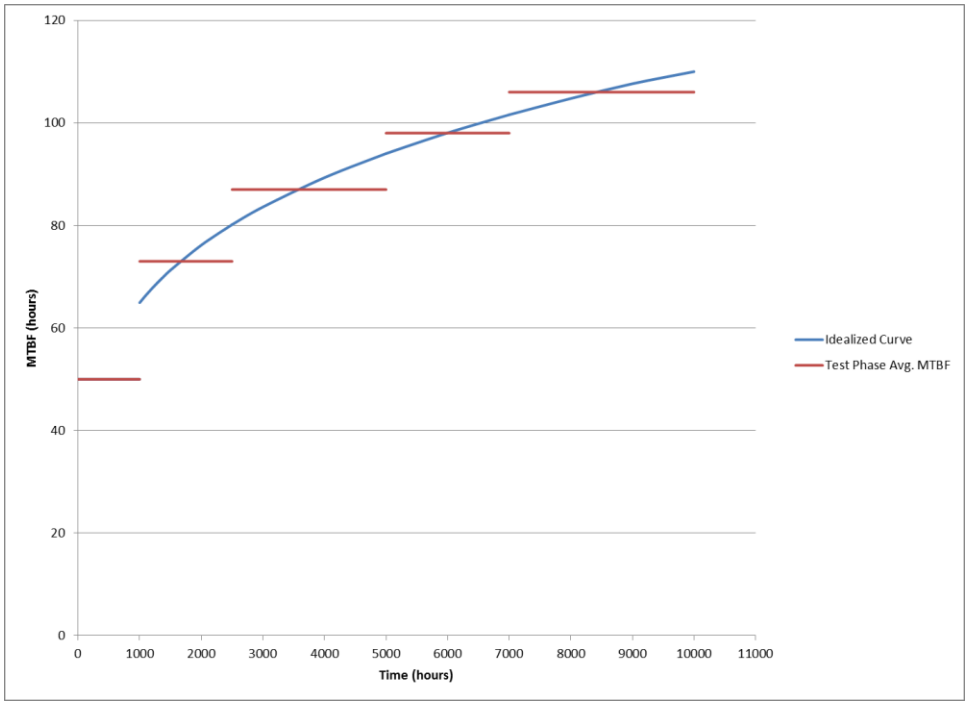


Figure 3.6.2.1-2. Example 2 Idealized Curve and Average Test Phase MTBF

Example 3: How to Determine How Much Test Time is Needed

Suppose that the average MTBF over a first test phase of t_i = 700 hours is estimated to be 1 hour. With a growth parameter of α = 0.4, how many test hours are needed to grow the reliability to a goal of a 3-hour MTBF?

From the above, the cumulative test time, T, necessary to grow from a 1-hour MTBF to 3-hour MTBF needs to satisfy the equation:

$$\text{Log}(T)=\log(700)+(1/0.4)[\log(3)+\log(1-0.6)]=8.02$$

That is, T = 3,043 hours.

Crow/ASMSAA Model

Table 3.6.2.1-4 summarizes the purpose, assumptions, limitations and benefits of the Crow/AMSAA reliability growth model (also known as the Reliability Growth Tracking Model Continuous (RGTMC).

Table 3.6.2.1-4: Crow/AMSAA Model Attributes

Attribute	Crow/AMSAA (RGTMC)
Purpose	• Assess the improvement in the reliability, within a single test phase, of a system during development for which usage is measured on a continuous scale • May be utilized both if (a) failure times are known and (b) if failure times are only known within defined intervals (i.e., grouped data)
Assumptions	• Test time is continuous • Failures within a test phase are occurring according to a NHPP with Power Law MVF
Limitations	• The model will not fit the test data if large jumps in reliability occur as a result of the applied fix implementation strategy • The model will be inaccurate if the testing does not adequately reflect the OMS/MP • If a significant number of non-tactical fixes are implemented, the growth rate and associated system reliability will be correspondingly optimistic • With respect to contributing to the reliability growth of the system, the model does not take into account reliability improvements due to delayed corrective actions
Benefits	• Can gauge demonstrated reliability versus planned reliability • Can provide statistical point estimates and confidence intervals for MTBF and growth rate • Allows for statistical goodness-of-fit testing

AMSAA employs the Weibull process to model reliability growth during a development test phase. This model adequately represents the improvement in reliability during development for a wide variety of systems. It is applicable to systems for which usage is measured on a continuous scale, for example, time in hours or distance in miles. Also, for high reliability and a large number of trials, the model may be used for one-shot systems. Development test programs are generally conducted on a phase-by phase-basis. For each test phase, it is typical for the test data to be compiled and a reliability evaluation made. It is important to note that the Crow/AMSAA model (RGTMC) is designed for tracking the reliability within a test phase and not across test phases. This model evaluates the reliability growth that results from the introduction of design fixes into the system during test and not the reliability growth that may occur at the end of a test phase due to delayed fixes.

The Crow/AMSAA model (RGTMC) assumes that, within a test phase, failures are occurring according to a NHPP. It is further assumed that the failure rate or intensity of failures during the test phase can be represented by the Weibull function, $p(t) = \lambda\beta t^{\beta-1}$. Under this model, the inverse of the failure intensity is interpreted as the instantaneous MTBF of the system at time, t. When "t" corresponds to the total cumulative time for the system (T), m(t) represents the demonstrated MTBF or the MTBF of the system in its present configuration.

Table 3.6.2.1-5 summarizes the options, required inputs and calculated outputs associate with the use of the basic Crow/AMSAA reliability growth model.

Table 3.6.2.1-5: Crow/AMSAA Model Options, Required Inputs and Calculated Outputs

Model	Options	Required Inputs	Calculated Outputs
Crow/AMSAA (RGTMC)	• Assess the growth in the reliability of a system during development for which usage is measured on a continuous scale • **Option 1**: Time-terminated testing • **Option 2**: Grouped Data approach	Option 1: • Total Test Time • Total Number of Observed Failures • Cumulative Failure Time (at each failure) Option 2: • Total Test Time • Number of Groups/Intervals • The Start Time of each Group/Interval • The End Time of each Group/Interval • The Number of Observed Failures in each Group/Interval	Option 1: • Estimate of Reliability Growth Parameter • Estimate of Model Scale Parameter • Estimate of MTBF at time, t • Estimate of intensity function at time, t • Estimate of MTBF and intensity function at Total Test Time • Unbiased Estimate of Reliability Growth Parameter • Estimate of Growth Rate • Expected number of failures at time, t • LCBs for True MTBF at Total Test Time • Cramér-Von Mises Goodness-of-Fit Statistic Option 2: • Estimate of Reliability Growth Parameter • Estimate of model scale parameter • Expected Number of Failures for each Group/Interval • Expected Average MTBF for each Group/Interval • Estimate of Growth Rate • Estimate of MTBF and intensity function for the last Group/Interval • LCBs for True MTBF for last Group/Interval • Chi-Square Goodness-of-Fit Statistic

Table 3.6.2.1-6 provides a brief overview of the elements of the Crow/AMSAA (RGTMC) model.

Table 3.6.2.1-6: Characteristics of the Crow/AMSAA Reliability Growth Model

Parameter	Comments
Growth Rate (β) (Shape Parameter)	The growth rate is: $\beta = \frac{N}{\sum_{i=1}^{N} \ln\left(T / X_i\right)}$ where, N = number of recorded failures T = total test time (= X_N when the test ends in a failure) X_i = time at which each individual failure occurs
Cumulative Failure Rate (λ_{cum})	The cumulative failure rate represents the measured failure rate, over time, as failures occur. It is calculated using the formula: $\lambda_{cum} = \lambda T^{\beta-1}$ where, λ = estimate of the initial failure rate (scale parameter), calculated from the formula: $\lambda = \frac{N}{T^{\beta}}$
Cumulative MTBF ($MTBF_{cum}$)	The formula for cumulative MTBF is simply the inverse of the formula for the cumulative failure rate, i.e.: $MTBF_{cum} = \frac{1}{\lambda T^{\beta-1}}$
Instantaneous Failure Rate (λ_{inst}) (Failure Intensity Function)	The instantaneous failure rate (and MTBF, for that matter) is the mathematical representation of the failure rate (or MTBF) if all previous failure occurrences are corrected. Its formula is: $\lambda_{inst} = \lambda\beta T^{\beta-1}$ or $\beta\lambda_{cum}$
Instantaneous MTBF ($MTBF_{inst}$)	The formula for instantaneous MTBF is, again, the inverse of the formula for the instantaneous failure rate, i.e.: $MTBF_{inst} = \frac{1}{\lambda\beta T^{\beta-1}}$ or $\frac{1}{\beta MTBF_{cum}}$

Example for Individual Failure Time Data

The following example demonstrates the Crow/AMSAA (RGTMC) option for individual failure time data in which two prototypes of a system are tested concurrently with the incorporation of design changes. The first prototype is tested for 132.4 hours, and the second is tested for 167.6 hours for a total of 300 cumulative test hours. Table 3.6.2.1-7 shows the accumulated test time on each prototype and the corresponding cumulative test time at each failure occurrence. An asterisk denotes a system failure. There are a total of 27 failures. Although the occurrence of two failures at exactly 16.5 hours is not possible under the assumption of the Crow/AMSAA (RGTMC) model, such data can result from rounding and are computationally tractable using the statistical estimation procedures described for the model in MIL-HDBK-189A. Note that the data are from a time-terminated test.

Table 3.6.2.1-7: Test Data for the RGTMC Individual Failure Time Option

Failure Number	Prototype #1 Hours	Prototype #2 Hours	Cumulative Hours	Failure Number	Prototype #1 Hours	Prototype #2 Hours	Cumulative Hours
1	2.60*	0.00	2.60	15	60.5	37.6*	98.1
2	16.5*	0.00	16.5	16	61.9*	39.1	101.1
3	16.5*	0.00	16.5	17	76.6*	55.4	132.0
4	17.0*	0.00	17.0	18	81.1	61.1*	142.2
5	20.5	0.90*	21.4	19	84.1*	63.6	147.7
6	25.3	3.80*	29.1	20	84.7*	64.3	149.0
7	28.7	4.60*	33.3	21	94.6*	72.6	167.2

Failure Number	Prototype #1 Hours	Prototype #2 Hours	Cumulative Hours	Failure Number	Prototype #1 Hours	Prototype #2 Hours	Cumulative Hours
8	41.8*	14.7	56.5	22	104.8	85.9*	190.7
9	45.5*	17.6	63.1	23	105.9	87.1*	193.0
10	48.6	22.0*	70.6	24	108.8*	89.9	198.7
11	49.6	23.4*	73.0	25	132.4	119.5*	251.9
12	51.4*	26.3	77.7	26	132.4	150.1*	282.5
13	58.2*	35.7	93.9	27	132.4	153.7*	286.1
14	59.0	36.5*	95.5	END	132.4	167.6	300.0

By using the 27 failure times listed under the columns labeled "Cumulative Hours" in Table 3.6.2.1-7, and applying the equations presented in Table 3.6.2.1-6, the following estimates are obtained:

- The point estimate for the shape parameter, β, is 0.716
- The point estimate for the scale parameter, λ, is 0.454
- The estimated failure intensity at the end of the test, λ_{inst}, is 0.0645 failures per hour
- The estimated MTBF at the end of the 300-hour test is 15.5 hours.

As shown in Figure 3.6.2.1-3, superimposing a graph of the estimated intensity function (instantaneous failure rate), using the equation below, on a plot of the average failure rate (using six 50-hour intervals) reveals decreasing failure intensity indicative of reliability growth:

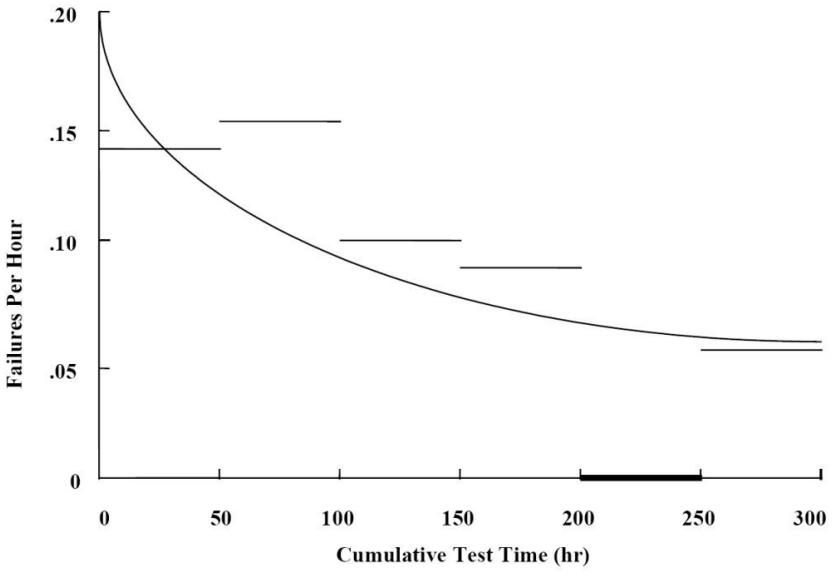


Figure 3.6.2.1-3: Estimated Intensity Function

Using Table 3.6.2.1-8 for the number of failures, F, equal to 27 and a statistical confidence level of 90 percent, the two-sided interval estimate for the MTBF at the end of the test is calculated as:

$MTBF_{Lower\ bound}$ = 0.636 * 15.5 hours = 9.9 hours
$MTBF_{Upper\ bound}$ = 1.682 * 15.5 hours = 26.1 hours

These results, and the estimated MTBF tracking growth curve (substituting "*t*" for "*T*" in the estimate of the MTBF equation presented earlier) are shown in Figure 3.6.2.1-4.

Table 3.6.2.1-8: Lower (L) and Upper (U) Coefficients for Confidence Intervals for MTBF from a Time-Terminated Reliability Growth Test

γ	0.80		0.90		0.95		0.98	
F	**L**	**U**	**L**	**U**	**L**	**U**	**L**	**U**
2	0.261	18.660	0.200	38.660	0.159	78.660	0.124	198.700
3	0.333	6.326	0.263	9.736	0.217	14.550	0.174	24.100
4	0.385	4.243	0.312	5.947	0.262	8.093	0.215	11.810
5	0.426	3.386	0.352	4.517	0.300	5.862	0.250	8.043
6	0.459	2.915	0.385	3.764	0.331	4.738	0.280	6.254
7	0.487	2.616	0.412	3.298	0.358	4.061	0.305	5.216
8	0.511	2.407	0.436	2.981	0.382	3.609	0.328	4.539
9	0.531	2.254	0.457	2.750	0.403	3.285	0.349	4.064
10	0.549	2.136	0.476	2.575	0.421	3.042	0.367	3.712
11	0.565	2.041	0.492	2.436	0.438	2.852	0.384	3.441
12	0.579	1.965	0.507	2.324	0.453	2.699	0.399	3.226
13	0.592	1.901	0.521	2.232	0.467	2.574	0.413	3.050
14	0.604	1.846	0.533	2.153	0.480	2.469	0.426	2.904
15	0.614	1.800	0.545	2.087	0.492	2.379	0.438	2.781
16	0.624	1.759	0.556	2.029	0.503	2.302	0.449	2.675
17	0.633	1.723	0.565	1.978	0.513	2.235	0.460	2.584
18	0.642	1.692	0.575	1.933	0.523	2.176	0.470	2.503
19	0.650	1.663	0.583	1.893	0.532	2.123	0.479	2.432
20	0.657	1.638	0.591	1.858	0.540	2.076	0.488	2.369
21	0.664	1.615	0.599	1.825	0.548	2.034	0.496	2.313
22	0.670	1.594	0.606	1.796	0.556	1.996	0.504	2.261
23	0.676	1.574	0.613	1.769	0.563	1.961	0.511	2.215
24	0.682	1.557	0.619	1.745	0.570	1.929	0.518	2.173
25	0.687	1.540	0.625	1.722	0.576	1.900	0.525	2.134
26	0.692	1.525	0.631	1.701	0.582	1.873	0.531	2.098
27	0.697	1.511	0.636	1.682	0.588	1.848	0.537	2.068
28	0.702	1.498	0.641	1.664	0.594	1.825	0.543	2.035
29	0.706	1.486	0.646	1.647	0.599	1.803	0.549	2.006
30	0.711	1.475	0.651	1.631	0.604	1.783	0.554	1.980
35	0.729	1.427	0.672	1.565	0.627	1.699	0.579	1.870
40	0.745	1.390	0.690	1.515	0.646	1.635	0.599	1.788
45	0.758	1.361	0.705	1.476	0.662	1.585	0.617	1.723
50	0.769	1.337	0.718	1.443	0.676	1.544	0.632	1.671
60	0.787	1.300	0.739	1.393	0.700	1.481	0.657	1.591
70	0.801	1.272	0.756	1.356	0.718	1.435	0.678	1.533
80	0.813	1.251	0.769	1.328	0.734	1.399	0.695	1.488
100	0.831	1.219	0.791	1.286	0.758	1.347	0.722	1.423

For F > 100,

$$L \approx \left(1 + z_{s+\gamma/2} \Big/ \sqrt{2F}\right)^{-2} \quad \text{and} \quad U \approx \left(1 - z_{s+\gamma/2} \Big/ \sqrt{2F}\right)^{-2}$$

where $z_{s+\gamma/2}$ is the 100 x $\left(0.5+\gamma/2\right)-th$ percentile of the Standard Normal distribution.

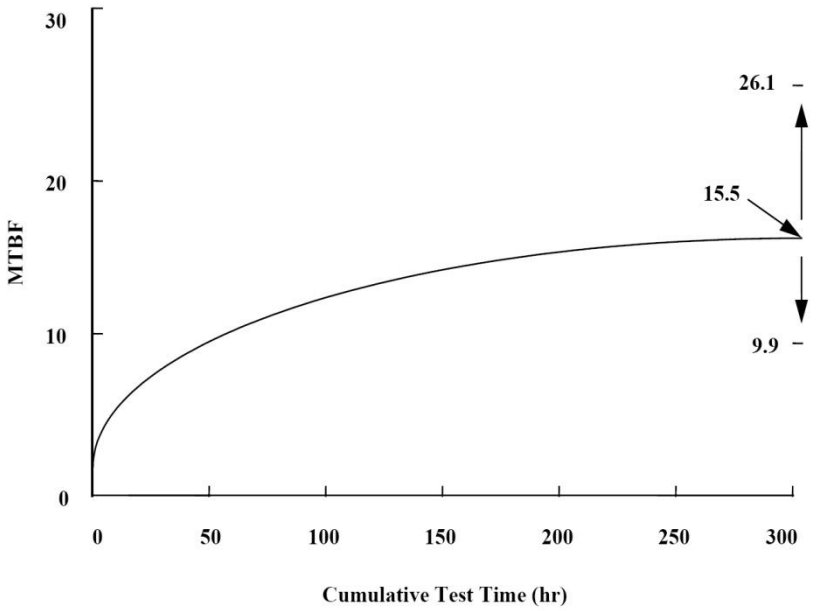


Figure 3.6.2.1-4: Estimated MTBF Function with 90% Interval Estimate at T = 300 Hours

Finally, to test the model goodness-of-fit, a Cramér-von Mises statistic is compared to the critical value from Table 3.6.2.1-9 corresponding to a chosen significance level of $\alpha = 0.05$ and the total observed number of failures of $F = 27$. Linear interpolation is used to arrive at the critical value.

The test statistic is calculated using the following equation:

$$C_M^2 = \frac{1}{12M} + \sum_{i=1}^{M}\left[\left(\frac{X_i}{X_N}\right)^{\bar{\beta}} - \frac{2i-1}{2M}\right]^2$$

where:

$$\bar{\beta} = \frac{N-2}{N}\hat{\beta}$$

Since the test statistic, 0.091, is less than the critical value, 0.218, we accept the hypothesis that the Crow/AMSAA model (RGTMC) is appropriate for this data set.

Table 3.6.2.1-9: Critical Values for the Cramer-Von Mises Goodness-of-Fit Test for Individual Failure Time Data

α / F	0.20	0.15	0.10	0.05	0.01
2	0.138	0.149	0.162	0.175	0.186
3	0.121	0.135	0.154	0.184	0.23
4	0.121	0.134	0.155	0.191	0.28
5	0.121	0.137	0.160	0.199	0.30
6	0.123	0.139	0.162	0.204	0.31
7	0.124	0.140	0.165	0.208	0.32
8	0.124	0.141	0.165	0.210	0.32
9	0.125	0.142	0.167	0.212	0.32
10	0.125	0.142	0.167	0.212	0.32

α / F	0.20	0.15	0.10	0.05	0.01
11	0.126	0.143	0.169	0.214	0.32
12	0.126	0.144	0.169	0.214	0.33
13	0.126	0.144	0.169	0.214	0.33
14	0.126	0.144	0.169	0.214	0.33
15	0.126	0.144	0.169	0.215	0.33
16	0.127	0.145	0.171	0.216	0.33
17	0.127	0.145	0.171	0.217	0.33
18	0.127	0.146	0.171	0.217	0.33
19	0.127	0.146	0.171	0.217	0.33
20	0.128	0.146	0.172	0.217	0.33
27	**0.128**	**0.146**	**0.172**	**0.218**	**0.33**
30	0.128	0.146	0.172	0.218	0.33
60	0.128	0.147	0.173	0.220	0.33
100	0.129	0.147	0.173	0.220	0.34

For F > 100, use values for F = 100; α = significance level

Figure 3.6.2.1-5 shows a graphical representation of a reliability growth plot based on MTBF.

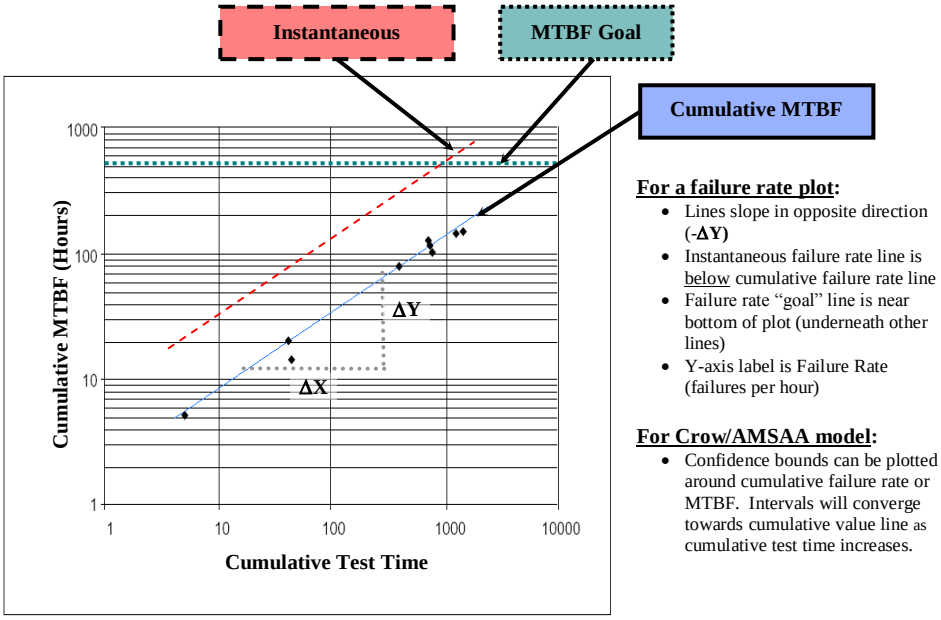


Figure 3.6.2.1-5: Typical Reliability Growth Plot (Duane)

Should there be an accept/reject criteria? The purpose of reliability growth testing is to uncover failures and take corrective actions to prevent their recurrence, resulting in a more robust design. Having an accept/reject criteria is a negative supplier incentive towards this purpose. Monitoring a supplier's progress and loosely defined thresholds are needed, but placing accept/reject criteria, or using a growth test as a demonstration, defeats the purpose of running them. The primary purpose of RGT is to improve the inherent design reliability, not evaluate or certify it.

How much validity/confidence should be placed on the numerical results of RGT? Associating a hard reliability estimate from a growth process, while mathematically practical, has the tone of an assessment process rather than an improvement process, especially if an RQT assessment will not follow the RGT. Use of the AMSAA

methodology provides the necessary statistical procedures for associating confidence levels with reliability results. In so doing, closer control over the operating conditions and failure determinations of the RGT must be exercised than if the test is for improvement purposes only. A better approach is to use a less closely controlled growth test as an improvement technique (or a structured extension of FRACAS, with greater emphasis on corrective action) to fine tune the design as insurance for an accept decision in an RQT.

The scope of the up-front reliability program, severity of the use environment and product state-of-the-art can have a large effect on the initial MTBF and, therefore, the test time required. The aggressiveness of the manufacturer in ensuring that fixes are developed and implemented can have a substantial effect on the growth rate and, therefore, test time. Other considerations for planning a growth test are provided in Table 3.6.2.1-10.

Table 3.6.2.1-10: RGT Planning Considerations

- To account for down time, calendar time should be estimated to be roughly twice the number of test hours
- A minimum test length of 5 times the predicted MTBF should always be used (if the Duane Model estimates less time). Literature commonly quotes typical test lengths of from 5 to 25 times the predicted MTBF
- For large MTBF systems (e.g., greater than 1000 hours), the preconditioning period equation does not hold; 250 hours is commonly used
- The upper limit on the reliability growth rate is 0.6 (reliability growth rates above 0.5 are rare). A growth rate of 0.25 to 0.4 is average for most projects (reference 8). A higher growth rate indicates that the effort to eliminate design weaknesses has been given top priority.

For More Information:

1. Crow, L.H., "An Extended Reliability Growth Model for Managing and Assessing Corrective Actions", Proceedings 2004 Annual Reliability and Maintainability Symposium
2. Crow, L.H., "The Extended Continuous Evaluation Reliability Growth Model", Proceedings 2010 Annual Reliability and Maintainability Symposium
3. Broemm, W. J., P. M. Ellner, and W. J. Woodworth, "AMSAA Reliability Growth Guide," AMSAA TR-652, U. S. Army Materiel Systems Analysis Activity, Aberdeen Proving Ground, MD 21005, September 2000
4. Coit, D.W., "Economic Allocation of Test Times for Subsystem-Level Reliability Growth Testing", IIE Transactions, No. 30, 1998, pp. 1143-1151
5. Lakey, P.B. and Neufelder, A.M., "System and Software Reliability Assurance Notebook", Rome Laboratory, RL-TR-97-XX, 1997
6. MIL-HDBK-189, "Reliability Growth Management", 13 February 1981 (MIL-HDBK-189C released 14 June 2011)
7. Nicholls, D., P. Lein, T. McGibbon, "Achieving System Reliability Growth Through Robust Design and Test", Reliability Information Analysis Center, 2011.
8. Benbow, D., H. Broome, "The Certified Reliability Engineer Handbook", ASQ Quality Press, 2010, ISBN 978-81-224-2792-9.

Topic 3.6.2.2: AMSAA Maturity Projection Model (AMPM)

A reliability projection is an assessment of reliability that can be anticipated at some future point in the development program given that corrective actions have been incorporated prior to that time. The AMSAA Maturity Projection Model (AMPM) is summarized in this section. A history of projection model development, as well as a more in depth discussion of AMPM is described in Reference 1 and in MIL-HDBK-189A.

Terminology that is pertinent to AMPM includes Management Strategy (MS), A-modes, B-modes, and Fix Effectiveness Factors (FEFs). All Defects are classified as A-mode or B-mode. B-mode defects are those for which corrective action will be developed during RGT, while A-mode defects are those which will not be addressed. The Management Strategy (MS) is a percentage reflective of the number of defects for which corrective action will be attempted compared to the total number of defects observed. The FEF is the percentage of a defect's failure rate that has been removed due to corrective action. For example, an FEF of 0.8 indicates that 80% of the failure rate associated with a defect has been removed as a result of corrective action. On this scale and FEF of 1.0 represents "perfect" corrective action, and an FEF of 0.0 represents completely ineffective corrective action.

A reliability projection is based on the reliability achieved to date through testing, analysis of test results, and engineering assessments of future program design and process characteristics. Projection is a particularly valuable analysis tool when a program is experiencing difficulties because it enables investigation of program alternatives. One can determine the reliability potential by performing "what-if" analyses on the Fix Effectiveness Factors (FEFs) for a proposed Management Strategy (MS). Projections can be used as a system or subsystem maturity metric, such as the initial failure rate surfaced. **Note, again, that the MS and FEFs can be very subjective, particularly in the absence of data or strong, documented rationale to support them. As a result, reliability projections may have little bearing on reality if the actual MS and FEFs, as implemented on a design or process, do not reflect the initial assumptions made.**

Extrapolating a reliability growth curve beyond the currently available data shows what reliability a program might be expected to achieve as a function of additional testing, provided the conditions of the test (i.e., the environmental and operational stresses) and the engineering effort to improve reliability (i.e., the MS and FEF processes) are maintained at their present levels (i.e., the current trend continues into the future). **The farther along the timeline the reliability is extrapolated, the higher the risk of a disconnect between the extrapolated and achieved reliability.**

Figure 3.6.2.2-1 provides a generic example of extrapolated and projected reliabilities.

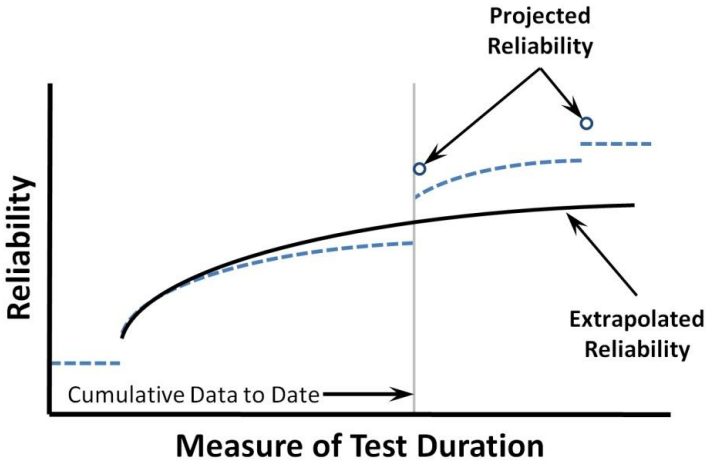


Figure 3.6.2.2-1: Extrapolated and Projected Reliabilities

The continuous version of the AMPM assumes that the test duration is measured on a continuous scale such as time, miles or cycles. Throughout this section, AMPM will refer to the continuous version of the model, and "time" will be the measure of test duration.

The AMPM addresses making reliability projections for several scenarios of interest. One case corresponds to that addressed by the ACPM, as discussed in Section 8.3 of Reference 1. This is the situation in which all fixes to B-modes are implemented at the end of the current test phase, Phase I, prior to commencing a follow-on test phase, Phase II. The projection problem is to assess the expected system failure intensity at the start of Phase II.

Another situation handled by the AMPM is the case where the reliability of the unit under test has been maturing over Test Phase I due to implemented fixes during Phase I. This case includes the situations where:

- All surfaced B-modes in Test Phase I have fixes implemented within this test phase, or
- Some of the surfaced B-modes are addressed by fixes within Test Phase I and the remainder are treated as delayed fixes, i.e., are fixed at the conclusion of Test Phase I, prior to commencing Test Phase II.

A third type of projection involves the system failure intensity at a future program milestone. This future milestone may occur beyond the commencement of the follow-on test phase.

All three types of projections are based on the Phase I B-mode first occurrence times, whether the associated B-mode fix is implemented within the current test phase or delayed (but implemented prior to the projection time). In addition to the B-mode first occurrence times, the projections are based on an average fix effectiveness factor (FEF). This average is with respect to all the potential B-modes, whether discovered or not (i.e., seen or unseen). However, as with the ACPM, this average FEF is determined based only on the seen (discovered) B-modes. For the AMPM model, the set of surfaced B-modes would typically be a mixture of B-modes addressed with fixes during the current test phase, as well as those addressed beyond the current test phase.

In some instances, a reliability projection for a future milestone can be based on extrapolating a reliability growth tracking curve. Such a curve only utilizes cumulative failure times and does not use B-Mode FEFs. This is a valid projection approach, provided it is reasonable to expect that the observed pattern of reliability growth will continue through the milestone of interest. However, this pattern could change in a pronounced manner. Reasons for such a change include:

- A change in the test environment
- A different level of future resources to analyze and implement effective corrective actions (invalidating initial FEF assumptions)
- Jumps in reliability due to delayed fixes

If extrapolating the current tracking curve is not deemed suitable due to these considerations, the AMPM projection methodology may be useful. Unlike assessments based on a tracking model, the AMPM assessments are independent of the fix discipline, as long as the fixes are implemented prior to the projection milestone date of interest. The AMPM (as well as the ACPM) utilizes a NHPP with regard to the number of distinct B-modes that occur over test duration, t. The associated pattern of B-mode first occurrence times is not dependent on the corrective action strategy, under the assumption that corrective actions are not inducing new B-modes to occur. Thus, the AMPM assessment procedure is not upset by jumps in reliability due to delayed groups of fixes. In contrast, reliability growth tracking curve methodology utilizes the pattern of cumulative failure times. Such a pattern is sensitive to the corrective action strategy. Thus, a reliability growth tracking curve model may not be appropriate for fitting failure data or for extrapolating due to a corrective action strategy that is not compatible with the model.

Note that AMPM reliability projections for a future milestone will be optimistic if corrective actions beyond the current test phase are less effective than the average FEF assessment based on B-modes discovered through the current test phase. Also, a change in the future testing environment could result in a new set of potential failure modes or affect the rates of occurrence of the original set. Either of these circumstances would tend to degrade the accuracy of the AMPM reliability projection.