NIST Special Publication 500-209

Computer Systems Technology

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology

NIST

Software Error Analysis

Wendy W. Peng
Dolores R. Wallace

NIST Special Publication 500-209

Software Error Analysis

Wendy W. Peng
Dolores R. Wallace

Systems and Software Technology Division
Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

April 1993

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

National Institute of Standards and Technology Special Publication 500-209
Natl. Inst. Stand. Technol. Spec. Publ. 500-209, 113 pages (Apr. 1993)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1993

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

ABSTRACT

This document provides guidance on software error analysis. Software error analysis includes error detection, analysis, and resolution. Error detection techniques considered in the study are those used in software development, software quality assurance, and software verification, validation and testing activities. These techniques are those frequently cited in technical literature and software engineering standards or those representing new approaches to support error detection. The study includes statistical process control techniques and relates them to their use as a software quality assurance technique for both product and process improvement. Finally, the report describes several software reliability models.

KEYWORDS

Data Collection; Error Detection; Error Removal; High Integrity Software; Metrics; Software Error Analysis; Software Quality Assurance; Software Verification and Validation; Statistical Process Control

EXECUTIVE SUMMARY

The main purpose of this document is to provide the software engineering community with current information regarding error analysis for software, which will assist them to do the following:

- Understand how error analysis can aid in improving the software development process;
- Assess the quality of the software, with the aid of error detection techniques;
- Analyze errors for their cause and then fix the errors; and
- Provide guidelines for the evaluation of high-integrity software.

The software industry is currently still young, without sufficient knowledge and adequate standards to guarantee fault-free software. Although research continues to identify better processes for error prevention, with current practices, errors will probably occur during software development and maintenance. Hence, there is the need for error analysis. Error analysis for software includes the activities of detecting errors, collecting and recording error data, analyzing and removing single errors, and analyzing collective error data to remove classes of errors. The collective error data may be used with statistical process control (SPC) techniques to improve the product and the processes used in developing, maintaining, and assuring the quality of software.

This report provides a description of error detection techniques which are cited frequently in technical literature and standards and describes the cost benefits of applying error detection early in the lifecycle. However, error detection alone is not sufficient for removal of an error. Information must be recorded about the error to assist in the analysis of its cause, its removal, and its relationship to the project and to other similar projects. This report provides guidance on data collection, analysis, and removal as well as error detection.

This report describes how several SPC techniques can be used for software quality assurance technique and for process improvement. The report identifies metrics related to software error detection and identifies several software reliability estimation models. Metrics are used to assess the product or process, while SPC techniques are used to monitor a project by observing trends. SPC techniques help to locate major problems in the development process, the assurance processes (e.g., software quality assurance, verification and validation), and the product itself.

The study of software engineering standards reported in [NUREG, NIST204] indicates that standards are beginning to associate requirements for error detection techniques with the quality requirements and problem types of the software project implementing the standard. Further examination of these documents and additional standards and guidelines for high integrity software indicates that these documents vary widely in their recommendations of specific error

techniques. Appendix B provides a summary of the error detection techniques required or recommended by these documents for the assurance of the quality of high integrity software.

This report recommends collection of error data into an organizational database for use by a vendor[1] over several projects, and modified collections of these databases for use by government auditors (e.g., Nuclear Regulatory Commission, Environmental Protection Agency). Software organizations should consider institutionalizing mechanisms for establishing and maintaining a database of error analysis data within their organization. Over time, it may become apparent that some error analysis techniques are more effective than others with respect to a given type of problem. It may also become apparent that problems in these areas occur most often with certain development practices and less frequently with others. The database must contain both developmental and operational error data to be effective. In the regulators' environment, auditors may use the information in the database to identify the most error-prone features of specific high integrity systems and may ensure that their audits examine these features carefully. The auditors may use the data to identify acceptance limits on different aspects of a high integrity system.

An organizational database may also play an important role in software reuse within an organization. In deciding whether or not to reuse a particular software component, one can examine its error history to determine whether it satisfies the level of assurance required by the intended application. One can evaluate the component by observing its past failure rates and fault densities to ensure that the component is appropriate for reuse. A software component may sometimes be reused to build a system which is of a higher level of assurance than that in which the component was originally used. The database would provide data on the reliability or other quality attributes to help determine how much additional work is needed to increase the quality of the component to the desired level.

[1]In this report, the term "vendor" includes software developers.

TABLE OF CONTENTS

TABLES

FIGURES

1. OVERVIEW

This document provides guidance on software error analysis. Error analysis includes the activities of detecting errors, of recording errors singly and across projects, and of analyzing single errors and error data collectively. The purpose of error analysis is to provide assurance of the quality of high integrity software.

The software industry is currently still young, without sufficient knowledge and adequate standards to guarantee fault-free software. Although research continues to identify better processes for error prevention, with current practices, errors will likely be entered into the software some time during development and maintenance. Hence, there is the need for error analysis, to aid in detecting, analyzing, and removing the errors.

The main purpose of this study is to provide the software engineering community with current information regarding error analysis, which will assist them to do the following:

- Understand how error analysis can aid in improving the software development process;
- Assess the quality of software, with the aid of error detection techniques;
- Analyze errors for their cause and then fix the errors; and
- Provide guidelines for the evaluation of high integrity software.

Section 2 discusses how error detection and analysis techniques can be used to improve the quality of software. Section 3 provides a global description of the principal detection techniques used in each software lifecycle phase and cost benefits for selected categories of these techniques. Section 4 provides guidance on collecting individual error data and removing single errors. Section 5 describes techniques for the collection and analysis of sets of error data, including statistical process control techniques and software reliability models. Section 6 provides a summary and recommendations based on this study of error analysis, and Section 7 provides a list of references. Appendix A contains detailed descriptions of common error detection techniques. Appendix B contains the results of a study of standards for high integrity software to determine the extent of coverage of error analysis techniques.

The error detection techniques and statistical techniques described in this report are a representative sampling of the most widely-used techniques and those most frequently referenced in standards, guidelines and technical literature. This report also describes the more common software reliability estimation models, most which are described in the American Institute of Aeronautics and Astronautics (AIAA) draft handbook for software reliability [AIAA]. Inclusion of any technique in this report does not indicate endorsement by the National Institute of Standards and Technology (NIST).

1.1. Definitions

Definitions of the following key terms used in this report are based on those in [IEEEGLOSS], [JURAN], [FLORAC], [SQE], [SHOOMAN], and [NIST204]. However, this report does not attempt to differentiate between "defect," "error," and "fault," since use of these terms within the software community varies (even among standards addressing these terms). Rather, this report uses those terms in a way which is consistent with the definitions given below, and with other references from which information was extracted.

anomaly. Any condition which departs from the expected. This expectation can come from documentation (e.g., requirements specifications, design documents, user documents) or from perceptions or experiences.
Note: An anomaly is not necessarily a problem in the software, but a deviation from the expected, so that errors, defects, faults, and failures are considered anomalies.

computed measure. A measure that is calculated from primitive measures.

defect. Any state of unfitness for use, or nonconformance to specification.

error. (1) The difference between a computed, observed, or measured value and the true, specified, or theoretically correct value or condition. (2) An incorrect step, process, or data definition. Often called a *bug*. (3) An incorrect result. (4) A human action that produces an incorrect result.
Note: One distinction assigns definition (1) to *error*, definition (2) to *fault*, definition (3) to *failure*, and definition (4) to *mistake*.

error analysis. The use of techniques to detect errors, to estimate/predict the number of errors, and to analyze error data both singly and collectively.

fault. An incorrect step, process, or data definition in a computer program. *See also*: **error**.

failure. Discrepancy between the external results of a program's operation and the software product requirements. A software failure is evidence of the existence of a fault in the software.

high integrity software. Software that must and can be trusted to work dependably in some critical function, and whose failure to do so may have catastrophic results, such as serious injury, loss of life or property, business failure or breach of security. Examples: nuclear safety systems, medical devices, electronic banking, air traffic control, automated manufacturing, and military systems.

primitive measure. A measure obtained by direct observation, often through a simple count (e.g., number of errors in a module).

primitive metric. A metric whose value is directly measurable or countable.

measure. The numerical value obtained by either direct or indirect measurement; may also be the input, output, or value of a metric.

metric. The definition, algorithm or mathematical function used to make a quantitative assessment of product or process.

problem. Often used interchangeably with **anomaly**, although **problem** has a more negative connotation, and implies that an error, fault, failure or defect does exist.

process. Any specific combination of machines, tools, methods, materials and/or people employed to attain specific qualities in a product or service.

reliability (of software). The probability that a given software system operates for some time period, without system failure due to a software fault, on the machine for which it was designed, given that it is used within design limits.

statistical process control. The application of statistical techniques for measuring, analyzing, and controlling the variation in processes.

2. INTRODUCTION TO SOFTWARE ERROR ANALYSIS

Software error analysis includes the techniques used to locate, analyze, and estimate errors and data relating to errors. It includes the use of error detection techniques, analysis of single errors, data collection, metrics, statistical process control techniques, error prediction models, and reliability models.

Error detection techniques are techniques of software development, software quality assurance (SQA), software verification, validation and testing used to locate anomalies in software products. Once an anomaly is detected, analysis is performed to determine if the anomaly is an actual error, and if so, to identify precisely the nature and cause of the error so that it can be properly resolved. Often, emphasis is placed only on resolving the single error. However, the single error could be representative of other similar errors which originated from the same incorrect assumptions, or it could indicate the presence of serious problems in the development process. Correcting only the single error and not addressing underlying problems may cause further complications later in the lifecycle.

Thorough error analysis includes the collection of error data, which enables the use of metrics and statistical process control (SPC) techniques. Metrics are used to assess a product or process directly, while SPC techniques are used to locate major problems in the development process and product by observing trends. Error data can be collected over the entire project and stored in an organizational database, for use with the current project or future projects. As an example, SPC techniques may reveal that a large number of errors are related to design, and after further investigation, it is discovered that many designers are making similar errors. It may then be concluded that the design methodology is inappropriate for the particular application, or that designers have not been adequately trained. Proper adjustments can then be made to the development process, which are beneficial not only to the current project, but to future projects.

The collection of error data also supports the use of reliability models to estimate the probability that a system will operate without failures in a specified environment for a given amount of time. A vendor[2] may use software reliability estimation techniques to make changes in the testing process, and a customer may use these techniques in deciding whether to accept a product.

The error data collected by a vendor may be useful to auditors. Auditors could request that vendors submit error data, but with the understanding that confidentiality will be maintained and that recriminations will not be made. Data collected from vendors could be used by the auditors to establish a database, providing a baseline for comparison when performing evaluations of high integrity software. Data from past projects would provide guidance to auditors on what to look for, by identifying common types of errors, or other features related to errors. For example, it could be determined whether the error rates of the project under evaluation are within acceptable bounds, compared with those of past projects.

[2]In this report, the term "vendor" includes software developers.

2.1. Cost Benefits of Early Error Detection

Ideally, software development processes should be so advanced that no errors will enter a software system during development. Current practices can only help to reduce the number of errors, not prevent *all* errors. However, even if the best practices were available, it would be risky to assume that no errors enter a system, especially if it is a system requiring high integrity.

The use of error analysis allows for early error detection and correction. When an error made early in the lifecycle goes undetected, problems and costs can accrue rapidly. An incorrectly stated requirement may lead to incorrect assumptions in the design, which in turn cause subsequent errors in the code. It may be difficult to catch all errors during testing, since exhaustive testing, which is testing of the software under all circumstances with all possible input sets, is not possible [MYERS]. Therefore, even a critical error may remain undetected and be delivered along with the final product. This undetected error may subsequently cause a system failure, which results in costs not only to fix the error, but also for the system failure itself (e.g., plant shutdown, loss of life).

Sometimes the cost of fixing an error may affect a decision not to fix an error. This is particularly true if the error is found late in the lifecycle. For example, when an error has caused a failure during system test and the location of the error is found to be in the requirements or design, correcting that error can be expensive. Sometimes the error is allowed to remain and the fix deferred until the next version of the software. Persons responsible for these decisions may justify them simply on the basis of cost or on an analysis which shows that the error, even when exposed, will not cause a critical failure. Decision makers must have confidence in the analyses used to identify the impact of the error, especially for software used in high integrity systems.

A strategy for avoiding the high costs of fixing errors late in the lifecycle is to prevent the situation from occurring altogether, by detecting and correcting errors as early as possible. Studies have shown that it is much more expensive to correct software requirements deficiencies late in the development effort than it is to have correct requirements from the beginning [STSC]. In fact, the cost to correct a defect found late in the lifecycle may be more than one hundred times the cost to detect and correct the problem when the defect was born [DEMMY]. In addition to the lower cost of fixing individual errors, another cost benefit of performing error analysis early in development is that the error propagation rate will be lower, resulting in fewer errors to correct in later phases. Thus, while error analysis at all phases is important, there is no better time, in terms of cost benefit, to conduct error analysis than during the software requirements phase.

2.2. Approach to Selecting Error Analysis Techniques

Planning for error analysis should be part of the process of planning the software system, along with system hazard analysis[3] and software criticality analysis. System hazard analysis is used to identify potential events and circumstances that might lead to problems of varying degrees of severity, from critical failures resulting in loss of life, to less serious malfunctions in the system. Software hazard analysis focuses on the role of the software relative to the hazards. Software criticality analysis may use the results of system and software hazard analyses to identify the software requirements (or design and code elements) whose erroneous implementation would cause the most severe consequences. Criticality analysis may also be used to identify project requirements that are essential to achieving a working software system. Critical software requirements are traced through the development process, so that developers can identify the software elements which are most error-prone, and whose errors would be catastrophic.

The results of hazard analysis and criticality analysis can be used to build an effective error analysis strategy. They aid in choosing the most appropriate techniques to detect errors during the lifecycle (see sec. 3). They also aid in the planning of the error removal process (i.e., the removal of individual errors, as described in sec. 4). Lastly, they aid in the selection of metrics, statistical process control techniques, and software reliability estimation techniques, which are described in section 5. Error analysis efforts and resources can be concentrated in critical program areas. Error analysis techniques should be chosen according to which type of errors they are best at locating. The selection of techniques should take into account the error profile and the characteristics of the development methodology. No project can afford to apply every technique, and no technique guarantees that every error will be caught. Instead, the most appropriate combination of techniques should be chosen to enable detection of as many errors as possible in the earlier phases.

[3]In this report, system hazard analysis may also include analysis of threats to security features of the software.

3. TECHNIQUES FOR DETECTING ERRORS

Software development and maintenance involves many processes resulting in a variety of products collectively essential to the operational software. These products include the statement of the software requirements, software design descriptions, code (source, object), test documentation, user manuals, project plans, documentation of software quality assurance activities, installation manuals, and maintenance manuals. These products will probably contain at least some errors. The techniques described in this section can help to detect these errors. While not all products are necessarily delivered to the customer or provided to a regulatory agency for review, the customer or regulatory agency should have assurance that the products contain no errors, contain no more than an agreed upon level of estimated errors, or contain no errors of a certain type.

This section of the report identifies classes of error detection techniques, provides brief descriptions of these techniques for each phase of the lifecycle, and discusses the benefits for certain categories of these techniques. Detailed descriptions of selected techniques appear in Appendix A. Detailed checklists provided in [NISTIR] identify typical problems that error detection techniques may uncover.

Error detection techniques may be performed by any organization responsible for developing and assuring the quality of the product. In this report, the term "developer" is used to refer to developers, maintainers, software quality assurance personnel, independent software verification and validation personnel, or others who perform error detection techniques.

3.1. Classes of Error Detection Techniques

Error detection techniques generally fall into three main categories of analytic activities: static analysis, dynamic analysis, and formal analysis. Static analysis is "the analysis of requirements, design, code, or other items either manually or automatically, without executing the subject of the analysis to determine its lexical and syntactic properties as opposed to its behavioral properties" [CLARK]. This type of technique is used to examine items at all phases of development. Examples of static analysis techniques include inspections, reviews, code reading, algorithm analysis, and tracing. Other examples include graphical techniques such as control flow analysis, and finite state machines, which are often used with automated tools. Traditionally, static analysis techniques are applied to the software requirements, design, and code, but they may also be applied to test documentation, particularly test cases, to verify traceability to the software requirements and adequacy with respect to test requirements [WALLACE].

Dynamic analysis techniques involve the execution of a product and analysis of its response to sets of input data to determine its validity and to detect errors. The behavioral properties of the program are also observed. The most common type of dynamic analysis technique is testing. Testing of software is usually conducted on individual components (e.g., subroutines, modules) as they are developed, on software subsystems when they are integrated with one another or with other system components, and on the complete system. Another type of testing is acceptance testing, often conducted at the customer's site, but before the product is accepted by the customer. Other examples of dynamic analyses include simulation, sizing and timing analysis, and prototyping, which may be applied throughout the lifecycle.

Formal methods involve rigorous mathematical techniques to specify or analyze the software requirements specification, design, or code. Formal methods can be used as an error detection technique. One method is to write the software requirements in a formal specification language (e.g., VDM, Z), and then verify the requirements using a formal verification (analysis) technique, such as proof of correctness. Another method is to use a formal requirements specification language and then execute the specification with an automated tool. This animation of the specification provides the opportunity to examine the potential behavior of a system without completely developing a system first.

3.2. Techniques Used During the Lifecycle

Criteria for selection of techniques for this report include the amount of information available on them, their citation in standards and guidelines, and their recent appearance in research articles and technical conferences. Other techniques exist, but are not included in this report. Tables 3-1a and 3-1b provide a mapping of the error detection techniques described in Appendix A to software lifecycle phases. In these tables, the headings R, D, I, T, IC, and OM represent the requirements, design, implementation, test, installation and checkout, and operation and maintenance phases, respectively. The techniques and metrics described in this report are applicable to the products and processes of these phases, regardless of the lifecycle model actually implemented (e.g., waterfall, spiral). Table B-2 in Appendix B lists which high integrity standards cite these error detection techniques.

Table 3-1a. Error Detection Techniques and Related Techniques (part 1)

TECHNIQUES	R	D	I	T	IC	OM
Algorithm analysis	■	■	■	■		■
Back-to-back testing				■		
Boundary value analysis				■		
Control flow analysis	■	■	■			■
Database analysis	■	■	■	■		■
Data flow analysis	■	■	■			■
Data flow diagrams	■					
Decision tables (truth tables)	■					

Table 3-1b. **Error Detection Techniques and Related Techniques (part 2)**

TECHNIQUES	R	D	I	T	I C	O M
Desk checking (code reading)			■			
Error seeding				■		
Finite state machines	■					
Formal methods (formal verification)	■	■				
Information flow analysis		■				
Inspections	■	■	■			
Interface analysis	■	■	■			
Interface testing				■		
Mutation analysis				■		
Performance testing				■		
Prototyping / animation	■	■	■			
Regression analysis and testing	■	■	■	■	■	■
Requirements parsing	■					
Reviews	■	■	■	■	■	■
Sensitivity analysis				■		
Simulation	■	■	■	■	■	■
Sizing and timing analysis		■	■	■		■
Slicing			■			
Software sneak circuit analysis			■			
Stress testing				■		
Symbolic evaluation			■			
Test certification				■	■	■
Tracing (traceability analysis)	■	■	■	■		
Walkthroughs	■	■	■	■	■	■

3.2.1. Requirements

During the requirements phase, static analysis techniques can be used to check adherence to specification conventions, consistency, completeness, and language syntax. Commonly used static analysis techniques during the requirements phase include control flow analysis, data flow analysis, algorithm analysis, traceability analysis, and interface analysis. Control and data flow analysis are most applicable for real time and data driven systems. These flow analyses employ transformation of text describing logic and data requirements into graphic flows which are easier to examine. Examples of control flow diagrams include state transition and transaction diagrams. Algorithm analysis involves rederivation of equations or the evaluation of the suitability of specific numerical techniques. Traceability analysis involves tracing the requirements in the software requirements specification to system requirements. The identified relationships are then analyzed for correctness, consistency, completeness, and accuracy. Interface analysis in this phase involves evaluating the software requirements specification with the hardware, user, operator, and software interface requirements for correctness, consistency, completeness, accuracy, and readability.

Dynamic analysis techniques can be used to examine information flows, functional interrelationships, and performance requirements. Simulation is used to evaluate the interactions of large, complex systems with many hardware, user, and other interfacing software components. Prototyping helps customers and developers to examine the probable results of implementing software requirements. Examination of a prototype may help to identify incomplete or incorrect requirements statements and may also reveal that the software requirements will not result in system behavior the customer wants. Prototyping is usually worthwhile when the functions of the computer system have not previously been used in automated form by the customer. In this case, the customer can change the requirements before costly implementation. Unless the project is small or an automated method can be used to build a prototype quickly, usually only carefully selected functions are studied by prototyping.

One approach for analyzing individual requirements is requirements parsing. This manual technique involves examination to ensure that each requirement is defined unambiguously by a complete set of attributes (e.g., initiator of an action, source of the action, the action, the object of the action, constraints). Because this technique identifies undefined attributes, it may prevent release of incomplete requirements to the designers. In those cases where the requirements are to be represented by a formal language specification, this analysis aids in clarifying a requirement before its transformation.

Languages based on formal methods, i.e., mathematically based languages, may be used to specify system requirements. The act of specifying the software requirements in a formal language forces reasoning about the requirements and becomes an error detection technique. When requirements have been written in a formal language, the task of simulation may be easier. Then, the behavior of the potential system can be observed through use of the simulation. It may be the combination of formal specifications with other error detection techniques (e.g., control flow analysis and data flow analysis) that provides the biggest payoff for using formal methods.

3.2.2. Design

Evaluation of the design provides assurance that the requirements are not misrepresented, omitted, or incompletely implemented, and that unwanted features are not designed into the product by oversight. Design errors can be introduced by implementation constraints relating to timing, data structures, memory space, and accuracy.

Static analysis techniques help to detect inconsistencies, such as those between the inputs and outputs specified for a high level module and the inputs and outputs of the submodules. The most commonly used static analysis techniques during this phase include algorithm analysis, database analysis, (design) interface analysis, and traceability analysis. As in the requirements phase, algorithm analysis examines the correctness of the equations and numerical techniques, but in addition, it examines truncation and rounding effects, numerical precision of word storage and variables (single vs. extended-precision arithmetic), and data typing influences. Database analysis is particularly useful for programs that store program logic in data parameters. Database analysis supports verification of the computer security requirement of confidentiality, by checking carefully the direct and indirect accesses to data. Interface analysis aids in evaluating the software design documentation with hardware, operator, and software interface requirements for correctness, consistency, completeness, and accuracy. Data items should be analyzed at each interface. Traceability analysis involves tracing the software design documentation to the software requirements documentation and vice versa.

Commonly used dynamic analysis techniques for this phase include sizing and timing analysis, prototyping, and simulation. Sizing and timing analysis is useful in analyzing real-time programs with response time requirements and constrained memory and execution space requirements. This type of analysis is especially useful for determining that allocations for hardware and software are made appropriately for the design architecture; it would be quite costly to learn in system test that the performance problems are caused by the basic system design. An automated simulation may be appropriate for larger designs. Prototyping can be used as an aid in examining the design architecture in general or a specific set of functions. For large complicated systems prototyping can prevent inappropriate designs from resulting in costly, wasted implementations.

Formal analysis involves tracing paths through the design specification and formulating a composite function for each, in order to compare these composite functions to that of the previous level. This process ensures that the design continues to specify the same functional solution as is hierarchically elaborated. This process can be applied manually, if the specification is sufficiently formal and exact, but is most feasible only for high level design specifications. However, with automated tools, the functional effects of all levels of the design can be determined, due to the speed and capacity of the tools for manipulating detailed specifications.

3.2.3. Implementation

Use of static analysis techniques helps to ensure that the implementation phase products (e.g., code and related documentation) are of the proper form. Static analysis involves checking that the products adhere to coding and documentation standards or conventions, and that interfaces and data types are correct. This analysis can be performed either manually or with automated tools.

Frequently used static analysis techniques during this phase include code reading, inspections, walkthroughs, reviews, control flow analysis, database analysis, interface analysis, and traceability analysis. Code reading involves the examination by an individual, usually an expert other than the author, for obvious errors. Inspections, walkthroughs, and reviews, which are all used to detect logic and syntax errors, are effective forerunners to testing. As in previous phases, control flow diagrams are used to show the hierarchy of main routines and their subfunctions. Database analysis is performed on programs with significant data storage to ensure that common data and variable regions are used consistently between all calling routines; that data integrity is enforced and no data or variable can be accidentally overwritten by overflowing data tables; and that data typing and use are consistent throughout the program. With interface analysis, source code is evaluated with the hardware, operator, and software interface design documentation, as in the design phase. Traceability analysis involves tracing the source code to corresponding design specifications and vice versa.

One category of static analysis techniques performed on code is complexity analysis. Complexity analysis measures the complexity of code based on specific measurements (e.g., number of parameters passed, number of global parameters, number of operands/operators). Although not an error detection technique, complexity analysis can be used as an aid in identifying where use of error detection techniques should be concentrated and also in locating test paths and other pertinent information to aid in test case generation.

Other static analysis techniques used during implementation which aid in error detection include software sneak circuit analysis and slicing. Software sneak circuit analysis is a rigorous, language-independent technique for the detection of anomalous software (i.e., "sneaks") which may cause system malfunction. The methodology involves creation of a comprehensive "pictorial" database using quasi-electrical symbology which is then analyzed using topological and application "clues" to detect faults in the code [PEYTON]. Slicing is a program decomposition technique used to trace an output variable back through the code to identify all code statements relevant to a computation in the program [LYLE]. This technique may be useful to demonstrate functional diversity.

Dynamic analysis techniques help to determine the functional and computational correctness of the code. Regression analysis is used to reevaluate requirements and design issues whenever any significant code change is made. This analysis ensures awareness of the original system requirements. Sizing and timing analysis is performed during incremental code development and analysis results are compared against predicted values.

A formal method used in the implementation phase is proof of correctness, which is applied to code.

3.2.4. Test

Dynamic analysis in the test phase involves different types of testing and test strategies. Traditionally there are four types of testing: unit, integration, system, and acceptance. Unit testing may be either structural or functional testing performed on software units, modules, or subroutines. Structural testing examines the logic of the units and may be used to support requirements for test coverage, that is, how much of the program has been executed. Functional testing evaluates how software requirements have been implemented. For functional testing, testers usually need no information about the design of the program because test cases are based on the software requirements.

Integration testing is conducted when software units are integrated with other software units or with system components. During integration testing, various strategies can be employed (e.g., top-down testing, bottom-up testing, sandwich testing) but may depend on the overall strategy for constructing the system. Integration testing focuses on software, hardware, and operator interfaces.

Both system testing and acceptance testing execute the complete system. The primary difference is that the developer conducts system testing, usually in the development environment, while the customer conducts acceptance testing (or commissions the developer to conduct the acceptance testing in the presence of the customer). Acceptance testing is supposed to occur in a fully operational customer environment, but in some cases (e.g., nuclear power plants, flight control systems), some parts of the environment may need to be simulated.

For all four types of testing, different strategies may be used, according to the project's characteristics. Some strategies include stress testing, boundary value testing, and mutation testing. Operational profile testing allows testers to select input data that are of special interest to the customer. For example, input data that causes execution of the most frequently used functions in operation may be the most important profile for testing for some systems. In other cases, it may be more important to choose an input profile that should not occur in reality. For nuclear power plants, this means choosing a profile that causes the software safety system to react; the system responses can be examined to determine system behavior in adverse circumstances.

A major problem with testing is knowing when to stop. Software reliability estimation techniques, such as those described in section 5 of this report, can be used to estimate the number of errors still present in the system, and to determine how much more testing is needed. Sensitivity analysis, a promising technique emerging from the research community and entering the marketplace, is intended to indicate where to test, and hence to determine how much to test [VOAS]. Because sensitivity analysis is derived from mutation testing which is intended for detecting small changes to a program, this technique depends on code that is already "close to correct" for its effectiveness.

3.2.5. Installation and Checkout

During this phase, it is necessary to validate that the software operates correctly with the operational hardware system and with other software, as specified in the interface specifications. It is also necessary to verify the correctness and adequacy of the installation procedures and certify that the verified and validated software is the same as the executable code approved for installation. There may be several installation sites, each with different parameters. It is necessary to check that the programs have been properly tailored for each site.

The most commonly used dynamic analysis techniques for this phase are regression analysis and test, simulation, and test certification. When any changes to the product are made during this phase, regression analysis is performed to verify that the basic requirements and design assumptions affecting other areas of the program have not been violated. Simulation is used to test operator procedures and to isolate installation problems. Test certification, particularly in critical software systems, is used to verify that the required tests have been executed and that the delivered software product is identical to the product subjected to software verification and validation (V&V).

3.2.6. Operation and Maintenance

During operation of an on-line continuous system, test cases may be constructed that will check periodically if the system is behaving as expected. For any software maintenance activity, error detection techniques should be selected as if the maintenance activity were a new development activity, but considering the impact of new changes to the system. Use of traceability analysis on the software products, including test documentation, is crucial to identifying the extent of use of any selected error detection technique on the total software system. Regression testing must be applied in this phase.

3.3. Benefits of Classes of Error Detection Techniques

In the early days of computers, static analysis of software involved hours of tedious manual checking of code for structural errors, syntax errors, and other types of errors. Today automation handles the tedious bookkeeping in both design and code activities. In the past, manual reading by an individual not only took longer, but may not always have been thorough. Design tools force consistency to some extent and support other static analyses. Techniques such as control flow analysis were difficult to perform manually, but with modern Computer Aided Software Engineering (CASE) tools, most static analyses can be performed more quickly and efficiently. As the power and ease of use of the tools improve, then static analyses become more effective.

A tool commonly used to perform static analysis is the compiler, which can detect syntactical code errors. The direct costs are the amount of electrical power and resources needed to conduct the compilation. However, not everyone agrees on the usefulness of compilers for producing error-free code. Supporters of Cleanroom engineering, a methodology for developing and assuring the quality of software, argue that the costs of rework and recompile are significant and should not be ignored. They believe that complete reliance on the tools to perform some of the intellectual work may reduce quality, because clean compilations can give a false sense of complete correctness [MILLS]. With Cleanroom engineering, programmers do not compile their code. Instead they spend more time on design, using a "box structure" method, and on analyzing their own work. When the programmers are confident of their work, it is submitted to another group who then compiles and tests the code. The Software Engineering Laboratory (SEL) at the National Aeronautics and Space Administration Goddard Space Flight Center collected sufficient data over 16 years to establish baselines for error and productivity rates. In recent years, two experiments were conducted on the Cleanroom approach [GREEN]. Results of the two Cleanroom experiments compared with SEL baselines show a lower error rate in the finished product and an increase in productivity across the lifecycle.

Software inspection, another static technique, is time consuming because it requires line by line, or graph by graph reading of the software element. Data collected from over 203 software inspections at the Jet Propulsion Laboratory in Pasadena, California, showed a significantly higher density of defects during requirements inspections [KELLY]. However, the defect densities of the products decreased exponentially from the requirements phase to the coding phase, implying that testing and rework will take less time. Code reading is another static analysis technique that has been shown in another SEL study to be quite effective [BASILI]. Researchers found that effectiveness increased with the experience level of the code readers, the reason being that experienced personnel were mentally executing the code. This technique may be difficult to schedule and implement formally; usually it is used when a programmer seeks help from a peer, and is conducted on small sections of code at a time. Also, errors found by code reading may not always be handled with a formal anomaly report.

Inspection and code reading have one drawback in common. The maximum benefits for these techniques are achieved when they are performed on all elements of the design and code, which tends to be time-consuming. Because of the time factor, they are usually conducted on only small portions of a program at a time, usually in three- or four-hour meetings. When the objective is to examine the entire program or many modules for global features, then other techniques with specific objectives (e.g., interface consistency, control flow, logic flow analysis) are more appropriate. Many of the static analysis techniques are intended to be conducted by individual members of a team, perhaps over days, and probably with automated support. There may be interaction among the team, especially to discuss possible anomalies. These types of techniques are effective for examining the integration of design or code modules.

Dynamic analyses tend to use large amounts of computer resources and involve human effort to prepare, execute and analyze tests. Testing can never guarantee that a system is completely correct, but it can demonstrate exactly what will occur under specific circumstances. Testing helps to establish an operational profile under which the system will work correctly. Testing also helps to uncover errors that were previously not discovered. Acceptance testing assures a customer that the software system will behave appropriately under specific circumstances as the customer has requested. Some CASE tools provide test aids (e.g., test case generators, test result capture, documentation). Although the power of modern computers has reduced execution time of test cases, nevertheless, exhaustive testing with all possible inputs under all circumstances is still not possible. In order to obtain maximum benefits from testing, careful planning and development of test goals, and strategies to achieve those goals are required.

While some static and dynamic analyses have become easier to perform with CASE tools, CASE technology has not eliminated all problems of software development and assurance. There are the problems of cost, methodology-dependence, and difficulty in understanding and using them. Two other major problems with CASE include restrictions on using the environments when they are not built on the concept of open systems [NIST187] and when information developed by the tools cannot be readily exchanged among tools of different vendors.

While journal articles and other literature describing usage of formal methods are available, they do not provide sufficient information to draw conclusions about the cost/quality benefits of using formal methods for the assurance of the quality of software. A study of formal methods was funded by NIST, the U.S. Naval Research Laboratory, and the Atomic Energy Control Board of Canada, to determine whether the benefits of using formal methods are significant relative to the costs of using them. The results of this study are published in [NISTGCR]. In the United Kingdom, there is an existing standard for safety critical software used in defense equipment, which requires the use of formal languages for specifications [MOD55].

The cost benefits of using specific error detection techniques or classes of techniques will differ from project to project. A balanced error detection program will depend on many factors, including the consequences of failure caused by an undetected error, the complexity of the software system, the types of errors likely to be committed in developing specific software, the effort needed to apply a technique, the automated support available, and the experience of the development and assurance staff. Another factor to consider is the interplay among techniques (e.g., whether the output of one technique can be used readily by another technique). If a specific error type is likely, then a technique known for finding that type of error should be selected. The application of formal verification techniques is appropriate when failure of the software would be disastrous. For planning a balanced program, an important requirement should be to ensure that analyses are applied to all the software products at all phases of the lifecycle in an orderly manner. The program should be evaluated frequently to ensure that the analyses are being used correctly and are aiding in error detection. The SPC techniques described in section 5 aid in this evaluation.

A final consideration for selecting techniques based on their cost benefit takes into account who will be conducting the analysis, and under what circumstances. For auditors, techniques which examine interfaces across the entire program, control flow, and critical operational paths are more appropriate than those involving detailed line by line analysis (e.g., software code inspection). When an anomaly is found, however, the auditors may choose to examine in greater detail the areas suspected of contributing to the anomaly.

4. REMOVAL OF ERRORS

This section describes the process of analyzing anomalies and removing errors. This is performed after an anomaly has been discovered using any error detection technique, such as those discussed in section 3. Analysis of an anomaly will not only aid in the removal of errors related to the anomaly, but will also help to detect other similar errors which have not yet manifested themselves. In addition, information obtained from this analysis can provide valuable feedback that may improve subsequent efforts and development processes in future projects.

The handling of an anomaly generally follows three steps: identification, investigation, and resolution. However, exact procedures for dealing with an anomaly will depend on many factors. First, it may be that the anomaly is not actually an error.[4] For example, the anomaly may be a result of misinterpreting test results. In these situations, an explanation about why the anomaly is not an error should be recorded, and no further action is required. Second, the procedures will depend on the activity used to detect the anomaly. For example, anomalies discovered during walkthroughs and code reading are often fixed immediately, without having to go through the formal error resolution process. During integration testing, all anomaly reports may be collected and then addressed to locate probable cause and recommend fixes. Third, the severity level of the anomaly will determine how soon the error should be fixed. Generally, the more severe the error, the sooner it needs to be fixed.

The general policy for handling anomalies should include rules/regulations concerning the administration of the entire error removal activity (e.g., who must fill out problem reports, where or to whom this information is distributed, how to close out problem reports, who enters the collected information into the error database). These issues are not addressed in this report, because the policy will be specific to an organization.

General project information which supports the error removal process should be maintained. This information may include, but is not limited to, descriptions of the design methodology, the verification plan used in design, the test plan, the configuration control plan, identification of tools used to design and test software (e.g., CASE tools), and the programming language used.

4.1. Identification

As soon as an anomaly is detected, information about it should be recorded to help identify, analyze, and correct the anomaly. Typically, this information is presented in an anomaly, or problem report. While the formats may differ, reports should include the following types of information.

[4]Or, the anomaly may be caused by a problem external to the software under analysis (e.g., the modem used for testing was not configured properly), not by an error in the software. In this case, the information on the anomaly is sent to the responsible party, but is not further addressed by the error removal activity.

Locator. Identify the person(s) who discovered the anomaly including name, address, phone number, email address, fax number, and company identification.

Date and Time. Specify the date and time that the anomaly occurred and/or was discovered. Time can be specified by wall clock time, system time, or CPU time. For distributed systems, specify the time zone.

Activity. Identify the activity taking place at the time the anomaly was discovered. These activities include error detection activities employed during the development and release of a product, including static and dynamic analysis, review, inspection, audit, simulation, timing analysis, testing (unit, integration, system, acceptance), compiling/assembling, and walkthrough.

Phase Encountered. Identify the lifecycle phase in which the anomaly was encountered (e.g., requirements, design, implementation, test, installation and checkout, and operation and maintenance). If possible, specify the activity within the phase (e.g., during preliminary design in the design phase).

Operational Environment. Specify the hardware, software, database, test support software, platform, firmware, monitor/terminal, network, and peripherals being used.

Status of Product. Specify the effect of the problem on the product (e.g., unusable, degraded, affected, unaffected).

Repeatability. Determine if the anomaly is a one-time occurrence, intermittent, recurring, or reproducible.

Symptom. Describe the symptoms, which are indications that a problem exists (e.g., inconsistency between the design and the requirements, violation of a constraint imposed by the requirements, operating system crash, program hang-up, input or output problem, incorrect behavior or result, error message, inconsistent logic or behavior, infinite loop, and unexpected abort).

Location of Symptom. The location of the anomaly can be in the actual product (hardware, software, database, or documentation), the test system, the platform, or in any development phase product (e.g., specification, code, database, manuals and guides, plans and procedures, reports, standards/policies). Identify the documentation that was analyzed, or the code that was executed, the tools and documentation used to support the activity. Identify the specific location(s) where the anomaly is evident in documentation, or the test case in which the anomaly occurred.

Severity. Severity is a measure of the disruption an anomaly gives the user when encountered during operation of the product. Severity can be divided into several levels, with the highest level being catastrophic, and the lowest being at the annoyance level. A severity classification system should be tailored to particular systems or class of systems. The number of levels and the corresponding descriptions of the levels may vary. An example of a severity classification is given below:

Level 6	Critical. Major feature not working, system crashes, loss of data
Level 5	Serious. Impairment of critical system functions, no workaround
Level 4	Major. Workaround is difficult
Level 3	Moderate. Workaround is simple
Level 2	Cosmetic. Tolerable, or fix is deferrable
Level 1	User misunderstanding
Level 0	No problem (e.g., testing error)

4.2. Investigation

Following the identification stage, all errors should be investigated to obtain further information on the nature and cause in order to propose solutions for resolution action or corrective action. Information that should be recorded during this stage include the following:

Phase Introduced. Identify the lifecycle phase in which the error was introduced. If possible, specify the activity within the phase (e.g., during preliminary design in the design phase).

Type. This attribute describes the type of error found, e.g., logic error, computational error, interface/timing error, data handling error, data error, documentation error, document quality error (e.g., out of date, inconsistent, incomplete), and enhancement errors (e.g., errors caused by change in requirements, errors caused by a previous fix).

Location of Error. The location of the error may be the same as the location of the symptom. See *Location of Symptom* in section 4.1.

Cause. Typical causes of an error include human errors (e.g., misunderstanding, omission errors) and inadequate methodology (e.g., incomplete inspection process, inappropriate design methodology).

Units Affected. This attribute identifies the software unit(s) affected by the error and its related fix, e.g., which components, modules, or documents are affected.

Priority. Priority is the degree of importance that is given to the resolution of an error. Based on the priority level, it is determined whether the error should be fixed immediately, eventually, or not at all (e.g., if error becomes obsolete as result of other changes). However, fixes should be performed according to the software configuration management policies of the project. The relationship between the priority scale and the severity scale should be specified by the administrative policy. An example of a priority scale is:

Level 5	Resolve error immediately
Level 4	Error gets high attention
Level 3	Error will be placed in normal queue
Level 2	Use workaround or fix in the interim
Level 1	Will be fixed last
Level 0	Will not be fixed

4.3. Resolution

Error resolution consists of the steps to correct the error. The policy of the project determines if the person who investigates the error will also correct the error. The procedures for distribution and retention of the error information is also identified by the policy. Typically, the recipients of the error information are the project manager, SQA manager, corporate database manager, and the customer. The amount of formalism (e.g., whether the plan needs to be documented) depends on the scope, risk, and size of the project. For small errors in small projects, this scheme may not be necessary.

4.3.1. Resolution Plan

The proposed procedures for resolution action should be documented in a resolution plan.

Item to Fix. Identify the item to be fixed (e.g., name, ID number, revision), the component within the item, text description of the fix.

Estimated Date or Time. Specify the proposed dates for start and completion of the fix.

Personnel. Identify the manager, engineer, or other members responsible for performing the fix and for follow-up.

4.3.2. Resolution Action

The resolution action is the actual fix, i.e., making changes in the product to correct and remove the error. The following information should be provided by the person(s) who perform the resolution action, upon completion.

Date Completed. Specify the date when resolution action (fix) was completed.

Personnel. Identify the person(s) who fixed the error.

Time Length. Specify the number of minutes or hours required for the fix.

Size of Fix. Specify the size of the fix in terms of the number of source lines of code (SLOC) added / modified, or the number of document pages added / modified.

4.3.3. Corrective Action

The corrective action stage is optional, because not all anomalies will require individual corrective actions.

Standards, Policies or Procedures. Identify the standards, policies, or procedures to be revised, created, or reinforced.

Other Action. This includes other revisions to the development process (e.g., implementing training, reallocation of people or resources, and improving or enforcing audit activities).

4.3.4. Follow-up

For all errors, there should be a follow-up to verify that the necessary resolution action or corrective action has been performed correctly.

Personnel. Identify the person or organization that performed follow-up of resolution action and/or corrective action.

Verification of Action. Confirm that the "right fix" has been performed, that the error has actually been removed, and that the fix has not adversely affected other areas of the software.

Disposition. Describe the status of the anomaly, whether it is closed (resolution was implemented and verified, or not within scope of project), deferred, merged with another problem, or referred to another project.

Closeout. Identify procedures for retention of the error data.

4.4. Use of Individual Error Data

The data that is collected for the purpose of removing a single error can be used for other purposes. This data can aid in removing all errors similar to the original error. In addition to making improvements in the product, data on single errors can be used to improve the current development process. For instance, if many errors are found to be requirements errors, this may prompt a change to the requirements specification process. Data on single errors can also be used in measurement and statistical process control activities such as those discussed in section 5. For example, the data can be used to calculate measures or it can be used as input to control charts. Finally, individual error data can be entered into an error database, in order to maintain an error history of all projects in the organization.

5. TECHNIQUES FOR THE COLLECTION AND ANALYSIS OF ERROR DATA

Techniques for collecting and analyzing sets of error data during the lifecycle aid in understanding, evaluating and improving the development and maintenance process or aid in evaluating or estimating product quality. Software measures provide insights about both process and product. Measures may feed into statistical process control (SPC) techniques; SPC techniques may be used for both process and product evaluation. Software reliability estimation techniques are usually applied only to the product. Most of these techniques operate on error history profiles of error data discovered by error detection techniques.

This section addresses *only* the error aspects of these techniques. Other information may need to be collected when making major decisions to change a policy or development activity (e.g., project information, customer requirements, company policy, methodologies being used, tools being used, number of people using a certain technique). These types of information are not discussed in this report.

5.1. Error History Profile / Database

An error history profile is needed to perform error analysis effectively. An organizational database can be used both to track the status of a project and to track the use of error analysis techniques. Data collected for the purpose of resolving single errors (e.g., source, cause, type, severity), should be placed in the error database to enable the establishment of anomaly histories. Other data collected specifically for the purpose of measurement or statistical process control should also be entered into the database. The database serves as a baseline for validation as well as for improvement. Past mistakes can be avoided from lessons learned. Maintaining a database serves the following purposes:

- To identify which development processes work well (or poorly) for an application domain,
- To support process improvement for the current project as well as for future projects,
- To identify whether the processes are being applied properly (or at all),
- To identify error classes by cause,
- To estimate the error growth rate throughout development, and therefore to be able to adjust plans for assurance activities, and
- To measure the quality of the software at delivery.

Error data collected from an error detection technique in one phase can support process improvement in an earlier lifecycle phase (for future projects), as well as in a later phase. For example, in a presentation at a COMPASS Conference,[5] one panel member explained that analysis of the data collected from code inspections at his organization revealed that a high percentage of the code errors were the result of errors in the software requirements specification. In response to this finding, the organization began investigating the use of formal languages for specifying software requirements. This example demonstrates the necessity of collecting and analyzing data for both error removal and process improvement.

Data histories can help managers to recognize when there is a significant deviation from project plans during development. Past error data can be used to estimate the number of expected errors at different times in the development cycle. For instance, if the reported error count for a particular product was smaller than was expected, compared with similar past projects, this may suggest that the development team generated an unusually low number of errors. However, further investigation may reveal that the project was behind schedule, and to save time, planned inspections were not held. Thus, many existing errors remained undetected, so that the low error count did not reflect the true quality of the product. This example illustrates how a history profile enables an organization to recognize and correct a process problem to avoid delivering a product with residual errors.

5.2. Data Collection Process

This section describes the process of collecting error data for a specific purpose (e.g., to use with control charts). Some of the error data may include data previously collected during error detection. This data can be retrieved from the organizational database, or can be collected directly upon discovery or during resolution of an error. Data must be collected properly in order for any error analysis technique to be effective. The recommended steps of the data collection process are listed below [AIAA]:

1. Establish the objectives.

2. Prepare a data collection plan. The plan may include the following recommended elements:

   Data definition and type. Specify/define the data to be collected and the type (i.e., attribute or variable data). An attribute is a characteristic that an item may or may not possess. It is obtained by noting the presence or absence of a characteristic and counting occurrences of the characteristic with a specified unit. For example: a module may or may not contain a defect. This type of data takes on only discrete (integer) values. Variable data is obtained by recording a numerical value for each item observed. Variable data can be either continuous or discrete. Examples: cost of fixing an error (continuous), lines of code (discrete).

[5]Dr. John Kelly, of the Jet Propulsion Laboratory, was a member of the "Formal Methods in Industry" panel at the 1992 COMPASS Conference, held at NIST in Gaithersburg, Maryland on June 15-18, 1992. The panel discussion was not documented in the conference proceedings.

Analysis technique. Identify the technique requiring the data. Each technique has unique data requirements, so the technique to be used should be specified prior to data collection.

Measurement method. Measurements can be taken by equipment, observation, or selecting data from existing records. The reliability, determined by accuracy and precision, of the measurement method must be established prior to data collection.

Sampling Procedure. The data collection interval, amount of data to be collected, and the sampling method should be specified (e.g., random sampling using a random number table). When determining the data collection interval, issues such as process volume, process stability, and cost should be considered.

Personnel. Identify persons responsible for specific data collection tasks.

Forms for data reporting (e.g., electronic spreadsheet, paper forms, etc.).

Recording and processing of data. One method for processing data is *blocking*, the separating of data into potential comparison categories during the recording of data. Blocking can be accomplished by recording each category separately, or through labeling information that enables future sorting.

Monitoring. Describe how the data collection process is to be monitored.

3. Apply tools. Automated tools should be considered whenever possible, in order to minimize impact on the project's schedule. Factors to consider include the following: availability of the tool, reliability of the tool, cost of purchasing or developing the tool, and whether it can handle any necessary adjustments.

4. Provide training. Once tools and plans are in place, training should be provided to ensure that data collectors understand the purpose of the measurements and know explicitly what data is to be collected.

5. Perform trial run. A trial run of the data collection plan should be made to resolve any problems or misconceptions about the plan. This can save vast amounts of time and effort.

6. Implement the plan. Collect the data and review them promptly, so that problems can be resolved before the disappearance of information required to resolve them (e.g., if test results on a screen are not saved).

7. Monitor data collection. Monitor the process to ensure that objectives are met and that procedures are implemented according to the data collection plan.

8. Use the data. Use the data as soon as possible to achieve maximum benefit.

9. Provide feedback to all involved. Those involved need to know what impact their efforts had, and the end result. This will enable them to understand the purpose of their efforts and agree to undertake similar tasks in the future.

Additional recommendations for the data collection process are listed below [ROOK]:

- Data collection should be integrated into the development process (e.g., as part of the quality management system).

- Data collection should be automated whenever possible.

- Time-scales between data collection and data analysis should be minimized.

- Data should be treated as a company resource and facilities should be available to keep historical records of projects as well as to monitor current projects.

- The problem of motivating personnel to keep accurate records should not be underestimated. Proper training and quick analysis facilities are essential, but are not sufficient.

5.3. Metrics

Within the software engineering community, there is much confusion and inconsistency over the use of the terms metric and measure. In this report, a metric is defined to be the mathematical definition, algorithm, or function used to obtain a quantitative assessment of a product or process. The actual numerical value produced by a metric is a measure. Thus, for example, cyclomatic complexity is a metric, but the value of this metric is the cyclomatic complexity measure.

Data on individual errors (see sec. 4) can be used to calculate metrics values. Two general classes of metrics include the following:

> *management metrics*, which assist in the control or management of the development process; and
> *quality metrics*, which are predictors or indicators of the product qualities

Management metrics can be used for controlling any industrial production or manufacturing activity. They are used to assess resources, cost, and task completion. Examples of resource-related metrics include elapsed calendar time, effort, and machine usage. Typical metrics for software estimate task completion include percentage of modules coded, or percentage of statements tested. Other management metrics used in project control include defect-related metrics. Information on the nature and origin of defects are used to estimate costs associated with defect discovery and removal. Defect rates for the current project can be compared to that of past projects to ensure that the current project is behaving as expected.

Quality metrics are used to estimate characteristics or qualities of a software product. Examples of these metrics include complexity metrics, and readability indexes for software documents. The use of these metrics for quality assessment is based on the assumptions that the metric measures some inherent property of the software, and that the inherent property itself influences the behavioral characteristics of the final product.

Some metrics may be both management metrics and quality metrics, i.e., they can be used for both project control and quality assessment. These metrics include simple size metrics (e.g., lines of code, number of function points) and primitive problem, fault, or error metrics. For example, size is used to predict project effort and time scales, but it can also be used as a quality predictor, since larger projects may be more complex and difficult to understand, and thus more error-prone.

A disadvantage of some metrics is that they do not have an interpretation scale which allows for consistent interpretation, as with measuring temperature (in degrees Celsius) or length (in meters). This is particularly true of metrics for software quality characteristics (e.g., maintainability, reliability, usability). Measures must be interpreted relatively, through comparison with plans and expectations, comparison with similar past projects, or comparison with similar components within the current project. While some metrics are mathematically-based, most, including reliability models, have not been proven.

Since there is virtually an infinite number of possible metrics, users must have some criteria for choosing which metrics to apply to their particular projects. Ideally, a metric should possess all of the following characteristics:

- Simple - definition and use of the metric is simple
- Objective - different people will give identical values; allows for consistency, and prevents individual bias
- Easily collected - the cost and effort to obtain the measure is reasonable
- Robust - metric is insensitive to irrelevant changes; allows for useful comparison
- Valid - metric measures what it is supposed to; this promotes trustworthiness of the measure

Within the software engineering community, two philosophies on measurement are embodied by two major standards organizations. A draft standard on software quality metrics sponsored by the Institute for Electrical and Electronics Engineers Software Engineering Standards Subcommittee supports the single value concept. This concept is that a single numerical value can be computed to indicate the quality of the software; the number is computed by measuring and combining the measures for attributes related to several quality characteristics. The international community, represented by the ISO/IEC organization through its Joint Technical Committee, Subcommittee 7 for software engineering appears to be adopting the view that a range of values, rather than a single number, for representing overall quality is more appropriate.

5.3.1. Metrics Throughout the Lifecycle

Metrics enable the estimation of work required in each phase, in terms of the budget and schedule. They also allow for the percentage of work completed to be assessed at any point during the phase, and establish criteria for determining the completion of the phase.

The general approach to using metrics, which is applicable to each lifecycle phase, is as follows: [ROOK]

- Select the appropriate metrics to be used to assess activities and outputs in each phase of the lifecycle.
- Determine the goals or expected values of the metrics.
- Determine or compute the measures, or actual values.
- Compare the actual values with the expected values or goals.
- Devise a plan to correct any observed deviations from the expected values.

Some complications may be involved when applying this approach to software. First, there will often be many possible causes for deviations from expectations and for each cause there may be several different types of corrective actions. Therefore, it must be determined which of the possible causes is the actual cause before the appropriate corrective action can be taken. In addition, the expected values themselves may be inappropriate, when there are no very accurate models available to estimate them.

In addition to monitoring using expected values derived from other projects, metrics can also identify anomalous components that are unusual with respect to other components values in the same project. In this case, project monitoring is based on internally generated project norms, rather than estimates from other projects.

The metrics described in the following subsections are defined in [ROOK], [IEEE982.2] and [AIRFORCE], [SQE], and [ZAGE] and comprise a representative sample of management and quality metrics that can be used in the lifecycle phases to support error analysis. This section does not evaluate or compare metrics, but provides definitions to help readers decide which metrics may be useful for a particular application.

5.3.1.1. Metrics Used in All Phases

Primitive metrics such as those listed below can be collected throughout the lifecycle. These metrics can be plotted using bar graphs, histograms, and Pareto charts as part of statistical process control. The plots can be analyzed by management to identify the phases that are most error prone, to suggest steps to prevent the recurrence of similar errors, to suggest procedures for earlier detection of faults, and to make general improvements to the development process.

Problem Metrics

Primitive problem metrics.

Number of problem reports per phase, priority, category, or cause
Number of reported problems per time period
Number of open real problems per time period
Number of closed real problems per time period
Number of unevaluated problem reports
Age of open real problem reports
Age of unevaluated problem reports
Age of real closed problem reports
Time when errors are discovered
Rate of error discovery

Cost and Effort Metrics

Primitive cost and effort metrics.
Time spent
Elapsed time
Staff hours
Staff months
Staff years

Change Metrics

Primitive change metrics.

Number of revisions, additions, deletions, or modifications
Number of requests to change the requirements specification and/or design during lifecycle phases after the requirements phase

Fault Metrics

Primitive fault metrics. Assesses the efficiency and effectiveness of fault resolution/removal activities, and check that sufficient effort is available for fault resolution/removal.

Number of unresolved faults at planned end of phase
Number of faults that, although fully diagnosed, have not been corrected, and number of outstanding change requests

Number of requirements and design faults detected during reviews and walkthroughs

5.3.1.2. Requirements Metrics

The main reasons to measure requirements specifications is to provide early warnings of quality problems, to enable more accurate project predictions, and to help improve the specifications.

Primitive size metrics. These metrics involve a simple count. Large components are assumed to have a larger number of residual errors, and are more difficult to understand than small components; as a result, their reliability and extendibility may be affected.

Number of pages or words
Number of requirements
Number of functions

Requirements traceability. This metric is used to assess the degree of traceability by measuring the percentage of requirements that has been implemented in the design. It is also used to identify requirements that are either missing from, or in addition to the original requirements. The measure is computed using the equation: $RT = R1/R2 \times 100\%$, where R1 is the number of requirements met by the architecture (design), and R2 is the number of original requirements. [IEEE982.2]

Completeness (CM). Used to determine the completeness of the software specification during requirements phase. This metric uses eighteen *primitives* (e.g., number of functions not satisfactorily defined, number of functions, number of defined functions, number of defined functions not used, number of referenced functions, and number of decision points). It then uses ten *derivatives* (e.g., functions satisfactorily defined, data references having an origin, defined functions used, reference functions defined), which are derived from the primitives. The metric is the weighted sum of the ten derivatives expressed as $CM = \sum w_i D_i$, where the summation is from i=1 to i=10, each weight w_i has a value between 0 and 1, the sum of the weights is 1, and each D_i is a derivative with a value between 1 and 0. The values of the primitives also can be used to identify problem areas within the requirements specification. [IEEE982.2]

Fault-days number. Specifies the number of days that faults spend in the software product from its creation to their removal. This measure uses two primitives: the phase, date, or time that the fault was introduced, and the phase, date, or time that the fault was removed. The fault days for the ith fault, (FD_i), is the number of days from the creation of the fault to its removal. The measure is calculated as follows: $FD = \sum FD_i$.

This measure is an indicator of the quality of the software design and development process. A high value may be indicative of untimely removal of faults and/or existence of many faults, due to an ineffective development process. [IEEE982.2]

Function points. This measure was originated by Allan Albrecht at IBM in the late 1970's, and was further developed by Charles Symons. It uses a weighted sum of the number of inputs, outputs, master files and inquiries in a product to predict development size [ALBRECHT]. To count function points, the first step is to classify each component by using standard guides to rate each component as having low, average, or high complexity. The second basic step is to tabulate function component counts. This is done by entering the appropriate counts in the Function Counting Form, multiplying by the weights on the form, and summing up the totals for each component type to obtain the Unadjusted Function Point Count. The third step is to rate each application characteristic from 0 to 5 using a rating guide, and then adding all the ratings together to obtain the Characteristic Influence Rating. Finally, the number of function points is calculated using the equation

$$\text{Function pts.} = \underset{\text{point count}}{\text{Unadjusted function}} * (.65 + .01 * \text{Character Influence Rating})$$

[SQE]

5.3.1.3. Design Metrics

The main reasons for computing metrics during the design phase are the following: gives early indication of project status; enables selection of alternative designs; identifies potential problems early in the lifecycle; limits complexity; and helps in deciding how to modularize so the resulting modules are both testable and maintainable. In general, good design practices involve high cohesion of modules, low coupling of modules, and effective modularity. [ZAGE]

Size Metrics

Primitive size metrics. These metrics are used to estimate the size of the design or design documents.

Number of pages or words
DLOC (lines of PDL)
Number of modules
Number of functions
Number of inputs and outputs
Number of interfaces

(Estimated) number of modules (NM). Provides measure of product size, against which the completeness of subsequent module based activities can be assessed. The estimate for the number of modules is given by, NM = S/M, where S is the estimated size in LOC, M is the median module size found in similar projects. The estimate NM can be compared to the median number of modules for other projects. [ROOK]

Fault Metrics

Primitive fault metrics. These metrics identify potentially fault-prone modules as early as possible. [ROOK]

> Number of faults associated with each module
> Number of requirements faults and structural design faults detected during detailed design

Complexity Metrics

Primitive complexity metrics. Identifies early in development modules which are potentially complex or hard to test. [ROOK]

> Number of parameters per module
> Number of states or data partitions per parameter
> Number of branches in each module

Coupling. Coupling is the manner and degree of interdependence between software modules [IEEE982.2]. Module coupling is rated based on the type of coupling, using a standard rating chart, which can be found in [SQE]. According to the chart, *data coupling* is the best type of coupling, and *content coupling* is the worst. The better the coupling, the lower the rating. [SQE], [ZAGE]

Cohesion. Cohesion is the degree to which the tasks performed within a single software module are related to the module's purpose. The module cohesion value for a module is assigned using a standard rating chart, which can be found in [SQE]. According to the chart, the best cohesion level is *functional*, and the worst is *coincidental*, with the better levels having lower values. Case studies have shown that fault rate correlates highly with cohesion strength. [SQE], [ZAGE]

(Structural) fan-in / fan-out. Fan-in/fan-out represents the number of modules that *call/are called by* a given module. Identifies whether the system decomposition is adequate (e.g., no modules which cause bottlenecks, no missing levels in the hierarchical decomposition, no unused modules ("dead" code), identification of critical modules). May be useful to compute maximum, average, and total fan-in/fan-out. [ROOK], [IEEE982.2]

Information flow metric (C). Represents the total number of combinations of an input source to an output destination, given by, $C = C_i \times (\text{fan-in} \times \text{fan-out})^2$, where C_i is a code metric, which may be omitted. The product inside the parentheses represents the total number of paths through a module. [ZAGE]

Design Inspection Metrics

Staff hours per major defect detected. Used to evaluate the efficiency of the design inspection processes. The following primitives are used: time expended in preparation for inspection meeting (T1), time expended in conduct of inspection meeting (T2), number of major defects detected during the ith inspection (S_i), and total number of inspections to date (I). The staff hours per major defect detected is given below, with the summations being from i=1 to i=I.

$$M = \frac{\sum (T1 + T2)_i}{\sum S_i}$$

This measure is applied to new code, and should fall between three and five. If there is significant deviation from this range, then the matter should be investigated. (May be adapted for code inspections). [IEEE982.2]

Defect Density (DD). Used after design inspections of new development or large block modifications in order to assess the inspection process. The following primitives are used: total number of unique defects detected during the ith inspection or ith lifecycle phase (D_i), total number of inspections to date (I), and number of source lines of design statements in thousands (KSLOD). The measure is calculated by the ratio

$$DD = \frac{\sum D_i}{KSLOD}$$ where the sum is from i=1 to i=I.

This measure can also be used in the implementation phase, in which case the number of source lines of executable code in thousands (KSLOC) should be substituted for KSLOD. [IEEE982.2]

Test Related Metrics.

Test related primitives. Checks that each module will be / has been adequately tested, or assesses the effectiveness of early testing activities. [ROOK]

Number of integration test cases planned/executed involving each module
Number of black box test cases planned/executed per module
Number of requirements faults detected during testing (also re-assesses quality of requirements specification)

5.3.1.4. Implementation Metrics

Metrics used during the implementation phase can be grouped into four basic types: size metrics, control structure metrics, data structure metrics, and other code metrics.

Size Metrics

Lines of Code (LOC). Although lines of code is one of the most popular metrics, it has no standard definition.[6] The predominant definition for line of code is "any line of a program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line." [SQE] It is an indication of size, which allows for estimation of effort, timescale, and total number of faults. For the same application, the length of a program partly depends on the language the code is written in, thus making comparison using LOC difficult. However, LOC can be a useful measure if the projects being compared are consistent in their development methods (e.g., use the same language, coding style). Because of its disadvantages, the use of LOC as a management metric (e.g., for project sizing beginning from the requirements phase) is controversial, but there are uses for this metric in error analysis, such as to estimate the values of other metrics. The advantages of this metric are that it is conceptually simple, easily automated, and inexpensive. [SQE]

Halstead software science metrics. This set of metrics was developed by Maurice Halstead, who claimed they could be used to evaluate the mental effort and time required to create a program, and how compactly a program is expressed. These metrics are based on four primitives listed below:

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = total occurrences of operators
N_2 = total occurrences of operands

The program length measure, N, is the sum of N_1 and N_2. Other software science metrics are listed below. [SQE]

Vocabulary: $n = n_1 + n_2$
Predicted length: $N\char`^ = (n_1 * \log_2 n_1) + (n_2 * \log_2 n_2)$
Program volume: $V = N * \log_2 n$
Effort: $E = (n_1 N_2 N \log_2 n)/(2n_2)$
Time: $T = E/ß$; Halstead $ß=18$
Predicted number of bugs: $B = V/3000$

Control Structure Metrics

Number of entries/exits per module. Used to assess the complexity of a software architecture, by counting the number of entry and exit points for each module. The equation to determine the measure for the ith module is simply $m_i = e_i + x_i$, where e_i is

[6]The SEI has made an effort to provide a complete definition for LOC. See [PARK].

the number of entry points for the ith module, and x_i is the number of exit points for the ith module. [IEEE982.2]

Cyclomatic complexity (C). Used to determine the structural complexity of a coded module in order to limit its complexity, thus promoting understandability. In general, high complexity leads to a high number of defects and maintenance costs. Also used to identify minimum number of test paths to assure test coverage. The primitives for this measure include the number of nodes (N), and the number of edges (E), which can be determined from the graph representing the module. The measure can then be computed with the formula, $C = E - N + 1$. [IEEE982.2], [SQE]

Data Structure Metrics

Amount of data. This measure can be determined by primitive metrics such as Halstead's n_2 and N_2, number of inputs/outputs, or the number of variables. These primitive metrics can be obtained from a compiler cross reference. [SQE]

Live variables. For each line in a section of code, determine the number of live variables (i.e., variables whose values could change during execution of that section of code). The average number of live variables per line of code is the sum of the number of live variables for each line, divided by the number of lines of code. [SQE]

Variable scope. The variable scope is the number of source statements between the first and last reference of the variable. For example, if variable A is first referenced on line 10, and last referenced on line 20, then the variable scope for A is 9. To determine the average variable scope for variables in a particular section of code, first determine the variable scope for each variable, sum up these values, and divide by the number of variables [SQE]. With large scopes, the understandability and readability of the code is reduced.

Variable spans. The variable span is the number of source statements between successive references of the variable. For each variable, the average span can be computed. For example, if the variable X is referenced on lines 13, 18, 20, 21, and 23, the average span would be the sum of all the spans divided by the number of spans, i.e., $(4+1+0+1)/4 = 1.5$. With large spans, it is more likely that a far back reference will be forgotten. [SQE]

5.3.1.5. Test Metrics

Test metrics may be of two types: metrics related to test results or the quality of the product being tested; and metrics used to assess the effectiveness of the testing process.

Primitive defect/error/fault metrics. These metrics can be effectively used with SPC techniques, such as bar charts, and Pareto diagrams. These metrics can also be used to form percentages (e.g., percentage of logic errors = number of logic errors ÷ total number of errors).

Number of faults detected in each module
Number of requirements, design, and coding faults found during unit and integration testing
Number of errors by type (e.g., number of logic errors, number of computational errors, number of interface errors, number of documentation errors)
Number of errors by cause or origin
Number of errors by severity (e.g., number of critical errors, number of major errors, number of cosmetic errors)

Fault density (FD). This measure is computed by dividing the number of faults by the size (usually in KLOC, thousands of lines of code). It may be weighted by severity using the equation

$$FD_w = (W_1\ S/N + W_2\ A/N + W_3\ M/N)\ /\ \text{Size}$$

where N = total number of faults
S = number of severe faults
A = number of average severity faults
M = number of minor faults
W_i = weighting factors (defaults are 10, 3, and 1)

FD can be used to perform the following: predict remaining faults by comparison with expected fault density; determine if sufficient testing has been completed based on predetermined goals; establish standard fault densities for comparison and prediction. [IEEE982.2], [SQE]

Defect age. Defect age is the time between when a defect is introduced to when it is detected or fixed. Assign the numbers 1 through 6 to each of the lifecycle phases from requirements to operation and maintenance. The defect age is then the difference of the numbers corresponding to the phase introduced and phase detected. The average defect age = $\frac{\Sigma \text{ (phase detected - phase introduced)}}{\text{number of defects}}$, the sum being over all the defects. [SQE]

Defect response time. This measure is the time between when a defect is detected to when it is fixed or closed. [SQE]

Defect cost. The cost of a defect may be a sum of the cost to analyze the defect, the cost to fix it, and the cost of failures already incurred due to the defect. [SQE]

Defect removal efficiency (DRE). The DRE is the percentage of defects that have been removed during a process, computed with the equation:

$$DRE = \frac{\text{Number of defects removed}}{\text{Number of defects at start of process}} \times 100\%$$

The DRE can also be computed for each lifecycle phase and plotted on a bar graph to show the relative defect removal efficiencies for each phase. Or, the DRE may be computed for a specific process (e.g., design inspection, code walkthrough, unit test, six-month operation, etc.). [SQE]

PROCESS METRICS

Test case metrics

Primitive test case metrics.

Total number of planned white/black box test cases run to completion
Number of planned integration tests run to completion
Number of unplanned test cases required during test phase

Coverage metrics[7]

Statement coverage. Measures the percentage of statements executed (to assure that each statement has been tested at least once). [SQE]

Branch coverage. Measures the percentage of branches executed. [SQE]

Path coverage. Measures the percentage of program paths executed. It is generally impractical and inefficient to test all the paths in a program. The count of the number of paths may be reduced by treating all possible loop iterations as one path. [SQE] Path coverage may be used to ensure 100 percent coverage of critical (safety or security related) paths.

Data flow coverage. Measures the definition and use of variables and data structures. [SQE]

[7]Commercial tools are available for statement coverage, branch coverage, and path coverage, but only private tools exist for data flow coverage. [BEIZER] Coverage tools report the percentage of items covered and lists what is not covered. [SQE]

Test coverage. Measures the completeness of the testing process. Test coverage is the percentage of requirements implemented (in the form of defined test cases or functional capabilities) multiplied by the percentage of the software structure (in units, segments, statements, branches, or path test results) tested. [AIRFORCE]

Failure metrics

Mean time to failure (MTTF). Gives an estimate of the mean time to the next failure, by accurately recording failure times t_i, the elapsed time between the ith and the (i-1)st failures, and computing the average of all the failure times. This metric is the basic parameter required by most software reliability models. High values imply good reliability. [IEEE982.2]

Failure rate. Used to indicate the growth in the software reliability as a function of test time and is usually used with reliability models. This metric requires two primitives: t_i, the observed time between failures for a given severity level i, and f_i, the number of failures of a given severity level in the ith time interval. The failure rate $\lambda(t)$ can be estimated from the reliability function R(t), which is obtained from the cumulative probability distribution F(t) of the time until the next failure, using a software reliability estimation model, such as the non-homogeneous Poisson process (NHPP) or Bayesian type model. The failure rate is $\lambda(t) = -1/R(t)\left[\frac{dR(t)}{dt}\right]$ where $R(t) = 1 - F(t)$. [IEEE982.2]

Cumulative failure profile. Uses a graphical technique to predict reliability, to estimate additional testing time needed to reach an acceptable reliability level, and to identify modules and subsystems that require additional testing. This metric requires one primitive, f_i, the total number of failures of a given severity level i in a given time interval. Cumulative failures are plotted on a time scale. The shape of the curve is used to project when testing will be complete, and to assess reliability. It can provide an indication of clustering of faults in modules, suggesting further testing for these modules. A non-asymptotic curve also indicates the need for continued testing. [IEEE982.2]

5.3.1.6. Installation and Checkout Metrics

Most of the metrics used during the test phase are also applicable during installation and checkout. The specific metrics used will depend on the type of testing performed. If acceptance testing is conducted, a requirements trace may be performed to determine what percentage of the requirements are satisfied in the product (i.e., number of requirements fulfilled divided by the total number of requirements).

5.3.1.7. Operation and Maintenance Metrics

Every metric that can be applied during software development may also be applied during maintenance. The purposes may differ somewhat. For example, requirements traceability may be used to ensure that maintenance requirements are related to predecessor requirements, and that the test process covers the same test areas as for the development. Metrics that were used during development may be used again during maintenance for comparison purposes (e.g., measuring the complexity of a module before and after modification). Elements of support, such as customer perceptions, training, hotlines, documentation, and user manuals, can also be measured.

Change Metrics

<u>Primitive change metrics</u>.

Number of changes
Cost/effort of changes
Time required for each change
LOC added, deleted, or modified
Number of fixes, or enhancements

Customer Related Metrics

<u>Customer ratings.</u> These metrics are based on results of customer surveys, which ask customers to provide a rating or a satisfaction score (e.g., on a scale of one to ten) of a vendor's product or customer services (e.g., hotlines, fixes, user manual). Ratings and scores can be tabulated and plotted in bar graphs.

<u>Customer service metrics</u>.

Number of hotline calls received
Number of fixes for each type of product
Number of hours required for fixes
Number of hours for training (for each type of product)

5.4. Statistical Process Control Techniques

Statistical process control is the application of statistical methods to provide the information necessary to continuously control or improve processes throughout the entire lifecycle of a product [OPMC]. SPC techniques help to locate trends, cycles, and irregularities within the development process and provide clues about how well the process meets specifications or requirements. They are tools for measuring and understanding process variation and distinguishing between random inherent variations and significant deviations so that correct decisions can be made about whether to make changes to the process or product.

To fully understand a process, it is necessary to determine how the process changes over time. To do this, one can plot error data (e.g., total number of errors, counts of specific types of errors) over a period of time (e.g., days, weeks, lifecycle phases) and then interpret the resulting pattern. If, for instance, a large number of errors are found in a particular phase, an investigation of the activities in that phase or preceding ones may reveal that necessary development activities were omitted (e.g., code reviews were not conducted during the implementation phase). A plot of the sources of errors may show that a particular group is the most frequent source of errors. Further investigation may confirm that members of the group do not have sufficient experience and training. A plot of the number of specific types of errors may show that many errors are related to incorrect or unclear requirements specifications (e.g., requirements are written in a way that consistently causes misinterpretations, or they fail to list enough conditions and restrictions). This would indicate that the process of requirements specification needs to be modified.

There are several advantages to using SPC techniques. First, errors may be detected earlier or prevented altogether. By monitoring the development process, the cause of the error (e.g., inadequate standards, insufficient training, incompatible hardware) may be detected before additional errors are created. Second, using SPC techniques is cost-effective, because less effort may be required to ensure that processes are operating correctly than is required to perform detailed checks on all the outputs of that process. Thus, higher quality may be achieved at a lower development expense. Finally, use of SPC techniques provides quantitative measures of progress and of problems so less guesswork is required [DEMMY].

Despite the advantages, there are also several potential disadvantages. To be successful, SPC requires discipline, planning, continuous commitment to the timely solution of process problems, and frequent access to information from the software lifecycle process [DEMMY].

5.4.1. Control Chart

The primary statistical technique used to assess process variation is the control chart. The control chart displays sequential process measurements relative to the overall process average and control limits. The upper and lower control limits establish the boundaries of normal variation for the process being measured. Variation within control limits is attributable to random or chance causes, while variation beyond control limits indicates a process change due to causes other than chance -- a condition that may require investigation. [OPMC] The upper control limit (UCL) and lower control limit (LCL) give the boundaries within which observed fluctuations are typical and acceptable. They are usually set, respectively, at three standard deviations above and below the mean of all observations. There are many different types of control charts, pn, p, c, etc., which are described in Table 5-1. This section is based on [OPMC], [SMITH], [CAPRIO], and [JURAN].

Table 5-1. Types of Control Charts

TYPE	DESCRIPTION	IMPLEMENTATION
np	number of nonconforming units (e.g., number of defective units)	The number of units in each sample with the selected characteristic is plotted; sample size is constant.
p	fraction of nonconforming units (e.g., fraction of defective units)	For each sample, the fraction nonconforming, obtained by dividing the number nonconforming by the total number of units observed, is plotted; sample size can change.
c	number of nonconformities (e.g., number of errors)	For each sample, the number of occurrences of the characteristic in a group is plotted; sample size is constant.
u	number of nonconformities per unit (e.g., number of errors per unit)	For each sample, the number of nonconformities per unit, obtained by dividing the number of nonconformities by the number of units observed, is plotted; sample size can change.
X	single observed value	The value for each sample of size 1 is plotted.
XB	X-Bar	For each sample, the mean of 2 to 10 observations (4 or 5 are optimal) is plotted.
R	range	The difference between the largest and smallest values in each sample is plotted.
XM	median	The median of each sample is plotted.
MR	moving range	The difference between adjacent measurements in each sample is plotted.

Implementation

1. Identify the purpose and the characteristics of the process to be monitored.
2. Select the appropriate type of control chart based on the type of characteristic measured, the data available, and the purpose of the application.
3. Determine the sampling method (e.g., number of samples (n), size of samples, time frame).
4. Collect the data.
5. Calculate the sample statistics: average, standard deviation, upper and lower control limits.
6. Construct the control chart based on sample statistics.
7. Monitor the process by observing pattern of the data points and whether they fall within control limits.

Interpretation

The existence of *outliers*, or data points beyond control limits, indicates that non-typical circumstances exist. A *run*, or consecutive points on one side of the average line (8 in a row, or 11 of 12, etc.) indicates a shift in process average. A *sawtooth* pattern, which is a successive up and down trend with no data points near the average line, indicates overadjustment or the existence of two processes. A *trend*, or steady inclining or declining progression of data points represents gradual change in the process. A *hug*, in which all data points fall near the average line, may indicate unreliable data. A *cycle*, or a series of data points which is repeated to form a pattern, indicates a cycling process.

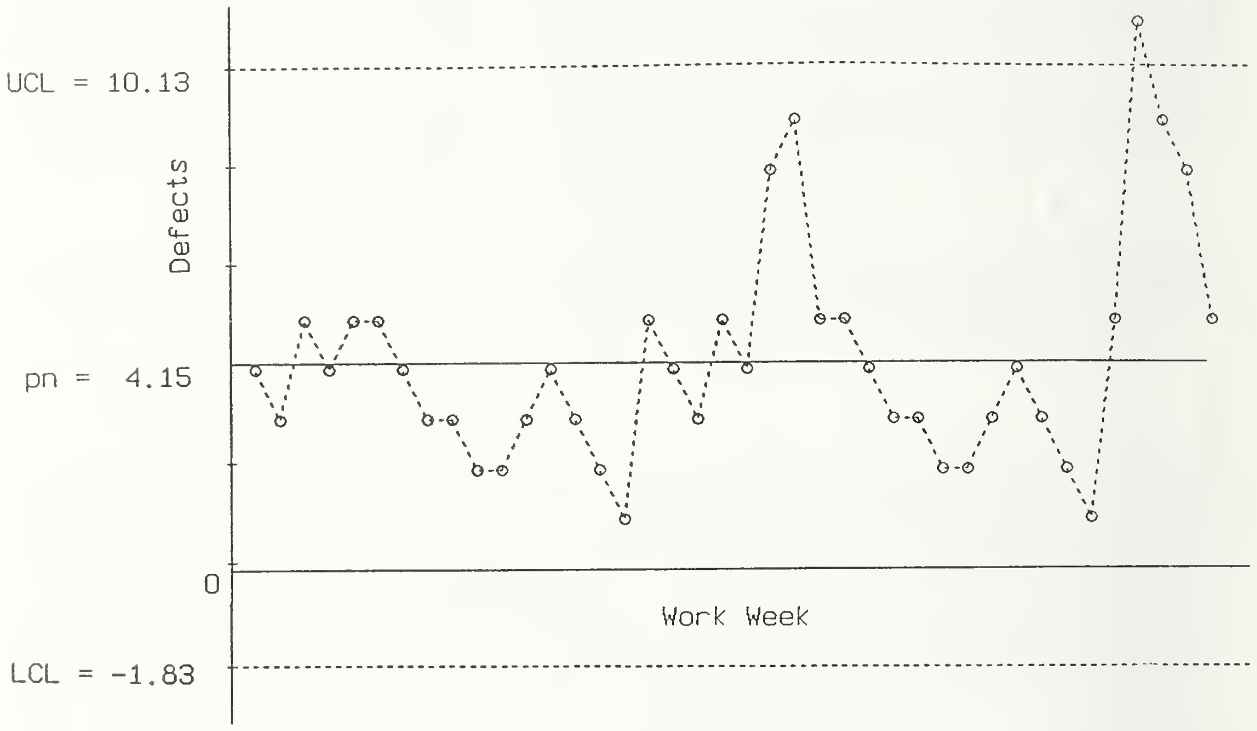

Figure 5-1. Example np Control Chart - Number of Defects Per Work Week.

Application Examples

Control charts are applicable to almost any measurable activity. Some examples for software include the following: number of defects/errors, training efforts, execution time, and number of problem reports per time period. An example of a np control with hypothetical data is shown in Figure 5-1. In this example, the number of samples (n) is 100. Each data point represents the number of defects found in the software product in a work week.

5.4.2. Run Chart

A run chart is a simplified control chart, in which the upper and lower control limits are omitted. The purpose of the run chart is more to determine trends in a process, rather than its variation. Although very simple, run charts can be used effectively to monitor a process, e.g., to detect sudden changes and to assess the effects of corrective actions. Run charts provide the input for establishing control charts after a process has matured or stabilized in time. Limitations of this technique are that it analyzes only one characteristic over time, and it does not indicate if a single data point is an outlier. This section is based on [OPMC] and [CAPRIO].

Implementation

1. Decide which outputs of a process to measure.
2. Collect the data.
3. Compute and draw the average line.
4. Plot the individual measurements chronologically.
5. Connect data points for ease of interpretation.

Interpretation

See Interpretation for Control Charts.

Application Examples

Run charts are applicable to almost any measurable activity. Some examples for software include the following: number of defects/errors, number of failures, execution time, and downtime.

5.4.3. Bar Graph

A bar graph is a frequency distribution diagram in which each bar represents a characteristic/ attribute, and the height of the bar represents the frequency of that characteristic. The horizontal axis may represent a continuous numerical scale (e.g., hours), or a discrete non-numerical scale (e.g., Module A, Module B or Requirements Phase, Design Phase). Generally, numerical-scale bar charts in which the bars have equal widths are more useful for comparison purposes; numerical-scale bar charts with unequal intervals can be misleading because the characteristics with the largest bars (in terms of area) do not necessarily have the highest frequency. This section is based on [SMITH].

Implementation

1. Define the subject and purpose.
2. Collect the data. Check that the sample size is sufficient.
3. Sort the data by frequency (or other measure) of characteristics.

4. For numerical-scale bar charts, determine the number of bars and the width of the bars (class width), by trying series of class widths, avoiding too fine or too coarse a granularity. The class widths in a chart may be all the same, or they may vary (as in fig. 5-2b), depending on how one wants to show the distribution of the data.
5. Construct the chart and draw the bars. The height of a bar represents the frequency of the corresponding characteristic.

Interpretation

In a simple bar graph in which the characteristics being measured are discrete and non-numerical (e.g., in fig. 5-2a) or if each bar has the same width, the measures for each characteristic can be compared simply by comparing the heights of the bars. For numerical-scale graphs with unequal widths, one can still compare the heights of the bars, but should remember not to interpret large bars as necessarily meaning that a large proportion of the entire population falls in that range.

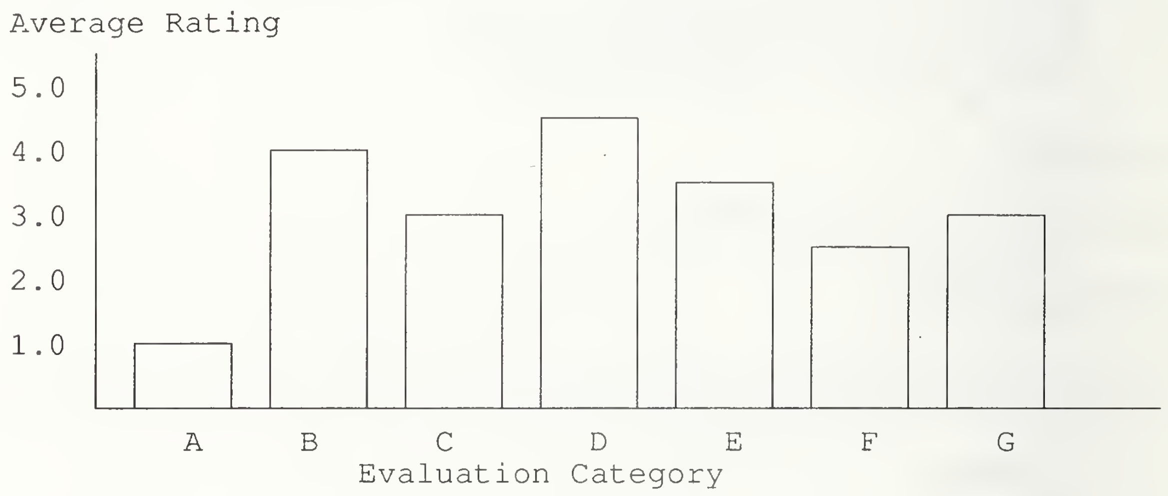

Figure 5-2a. Example Bar Chart - Customer Ratings (5 is best, 1 is worst).

Application Examples

Bar graphs are useful for analyzing and displaying many different types of data. It is mostly used to compare the frequencies of different attributes. For example, in Figure 5-2a, it is used to plot the average customer rating for each evaluation category (e.g., customer service, hotlines, overall satisfaction). The graph shows clearly that Category D has the highest rating. Figure 5-2b illustrates how numerical-scale bar charts can be used for software analysis. Based on hypothetical data, it shows the percentage of modules falling in each defect range. For instance, the graph shows that 30% of all modules contain 10-20 defects and 5% contain 20-25 defects. Other examples of characteristics that may be plotted include: number or percentage of errors by lifecycle phase, by type, or by cause, and number or percentage of problem reports by phase or by type. See also section 5.3.1.1 on primitive problem metrics for additional examples.

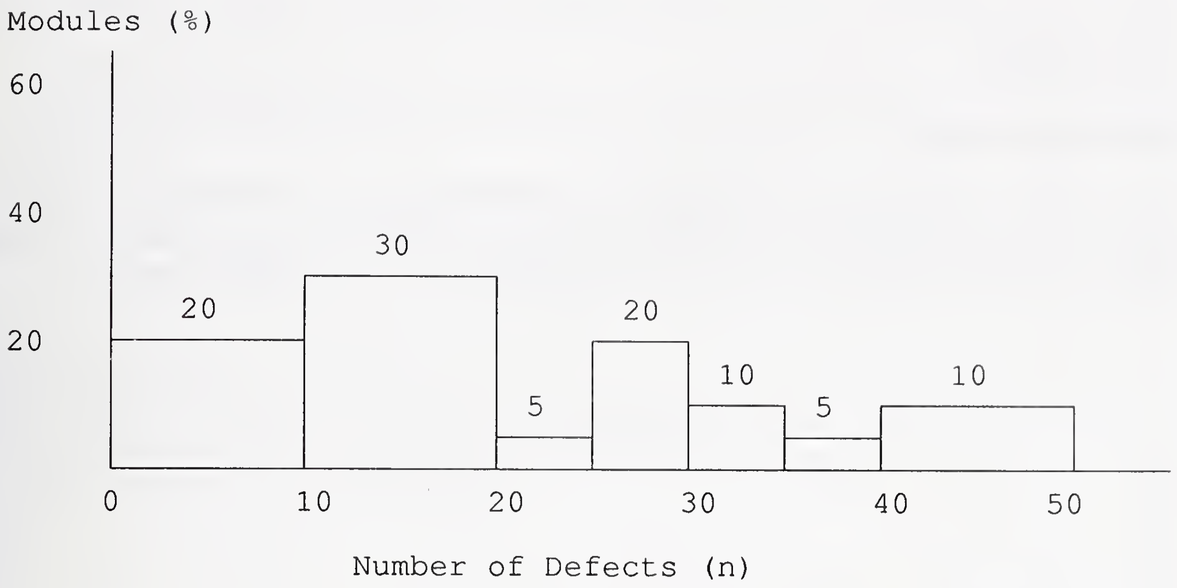

Figure 5-2b. Example Bar Chart - Number of Modules with *n* Defects.

5.4.4. Pareto Diagram

A Pareto diagram is a special use of the bar graph in which the bars are arranged in descending order of magnitude. The purpose of Pareto analysis, using Pareto diagrams, is to identify the major problems in a product or process, or more generally, to identify the most significant causes for a given effect. This allows a developer to prioritize problems and decide which problem area to work on first. This section is based on [OPMC] and [CAPRIO].

Implementation

1. Follow the steps for constructing a bar graph, except that the bars should be in descending order of magnitude (height).
2. Determine the "vital few" causes by drawing a cumulative percent line and applying the 20/80 rule.
3. Compare and identify the major causes. Repeat process until root cause of the problem is revealed.

Interpretation

Pareto analysis is based on the 20/80 rule, which states that approximately 20% of the causes (the "vital few") account for 80% of the effects (problems). The "vital few" can be determined by drawing a cumulative percent line and noting which bars are to the left of the point marking 80% of the total count. The vital few are usually indicated by significantly higher bars and/or a relatively steep slope of the cumulative percent line. In Figure 5-2, the vital few are logic, computational, and interface errors since 80% of the errors are found in these modules. By knowing the primary causes of a problem or effect, the developer can decide where efforts should be concentrated.

Application Examples

Most data that can be plotted on a non-numerical scale bar graph can also be plotted on a Pareto diagram. Examples include: number or percentage of errors by type, by cause, or by lifecycle phase, and number or percentage of problem reports by type or by lifecycle phase.

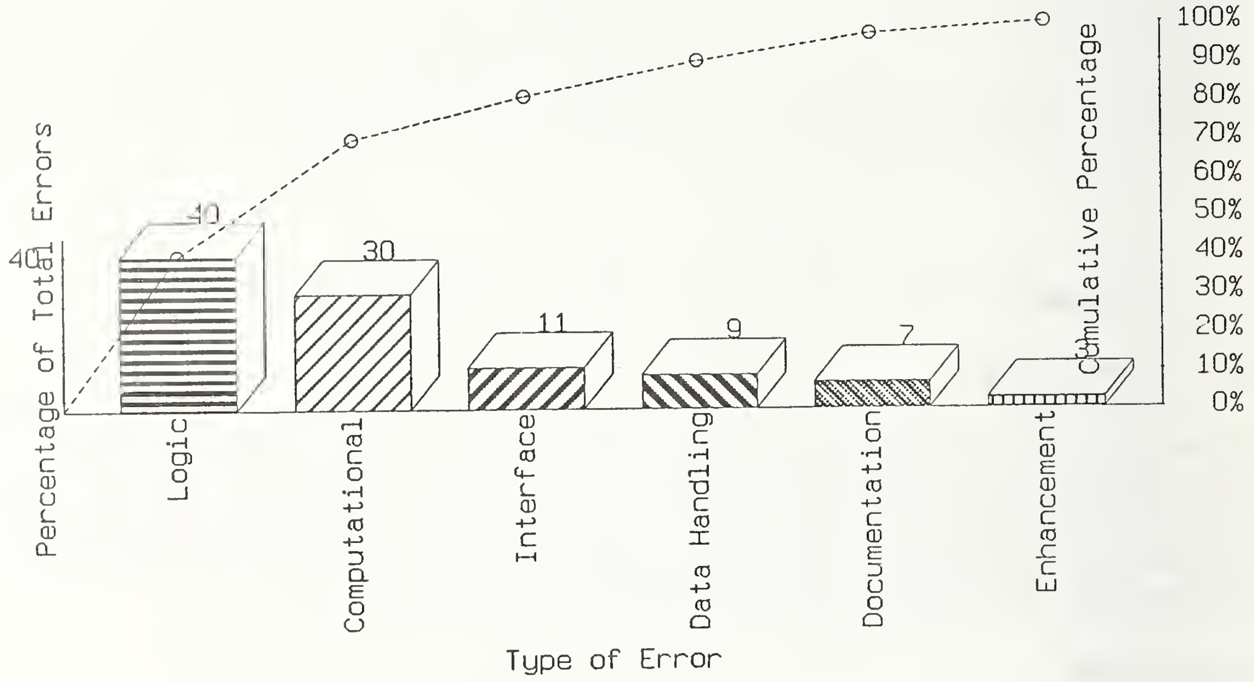

Figure 5-3. Example Pareto Chart - Percentage of Defects by Type.

5.4.4. Histogram

A histogram is a frequency distribution diagram in which the frequencies of occurrences of the different variables being plotted are represented by bars. The purpose is to determine the shape of the graph relative to the normal distribution (or other distributions). It is often confused with a bar graph, in which the frequency of a variable is indicated by the height of the bars. In a histogram, the frequency is indicated by the *area* of the bar. Histograms can only be used with variable data, which require measurements on a continuous scale. Only one characteristic can be shown per histogram, and at least 30 observations representing homogenous conditions are needed. This section is based on [OPMC], [CAPRIO], and [FREEDMAN].

Implementation

1. Define the subject and purpose.
2. Collect the data and organize from smallest to largest values. Check that sample size is sufficient.
3. Calculate the range (r), i.e. the difference between the largest and smallest values.
4. Decide arbitrarily on the number of bars (k), usually between 7 and 13.
5. To make bars of equal width, use the equation, $w = r/k$ to calculate the interval or width (w) of the bars.
6. Sort the data into the appropriate intervals and count the number of data points that fall in each interval.
7. Calculate the frequencies (actual counts or percent) for each interval.
8. Draw the bars. The height of the bar is calculated by dividing the frequency by w, the width of the bar (in horizontal units).

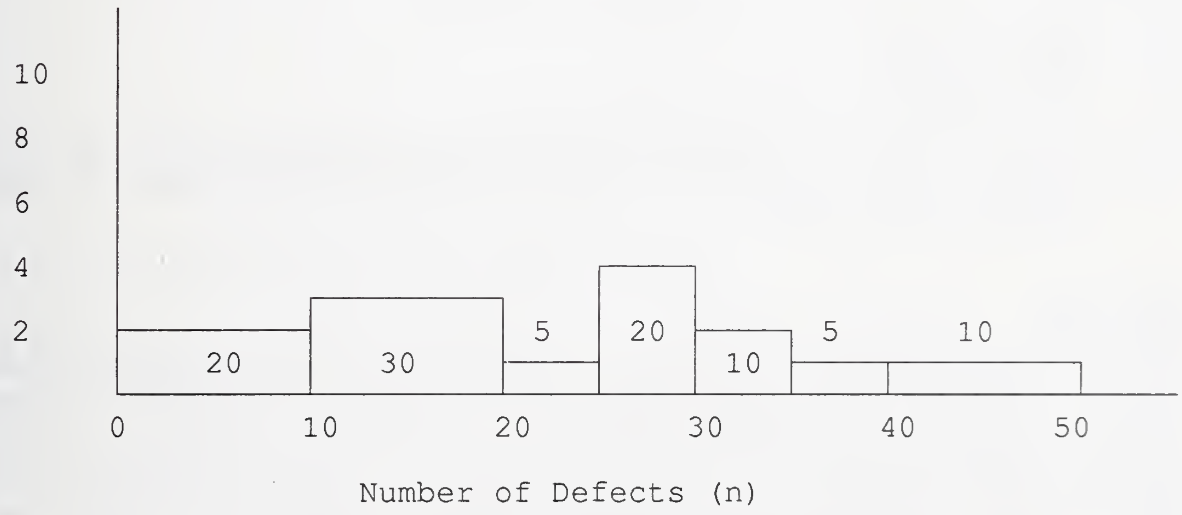

Figure 5-4. Example Histogram - Number of Modules with n Defects.

Interpretation

A histogram is a frequency distribution, in which the area of each bar is always proportional to the actual percentage of the total falling in a given range. For example, Figure 5-4 shows that 30% of all modules contain 10-20 defects, indicated by the largest bar. Both Figure 5-4 and Figure 5-2a are plotted with the same data. Note the difference in the relative size of the bars. If the bars are of equal width, then the histogram is equivalent to a bar graph, in which the relative size of the bars depends only on their heights. A histogram can be compared to the normal distribution (or other distribution). For example, if the graph is off-center or skewed, this may indicate that a process requires adjustment.

Application Examples

Histograms are essentially used for the same applications as bar graphs, except that the horizontal scale in a histogram must be numerical, usually representing a continuous random variable. See Application Examples for Bar Graphs.

5.4.5. Scatter Diagram

A scatter diagram is a plot of the values of one variable against those of another variable to determine the relationship between them. This technique was popularized by Walter Shewhart at Bell Laboratories. Scatter diagrams are used during analysis to understand the cause and effect relationship between two variables. They are also called correlation diagrams. This section is based on [KITCHENHAM], [OPMC], and [CAPRIO].

Implementation

1. Define the subject and select the variables.
2. Collect the data.
3. Plot the data points using an appropriate scale.
4. Examine the pattern to determine whether any correlation exists (e.g., positive, negative). For a more precise specification of the relationship, regression, curve fitting or smoothing techniques can be applied.

Interpretation

If the data points fall approximately in a straight line, this indicates that there is a linear relationship, which is positive or negative, depending on whether the slope of the line is positive or negative. Further analysis using the method of least squares can be performed. If the data points form a curve, then there is a non-linear relationship. If there is no apparent pattern, this may indicate no relationship. However, another sample should be taken before making such a conclusion.

Application Examples

The following are examples of pairs of variables that might be plotted:

- Complexity vs. defect density (example shown in fig. 5-5)
- Effort vs. duration (of an activity)
- Failures vs. time
- Failures vs. size
- Cost vs. time

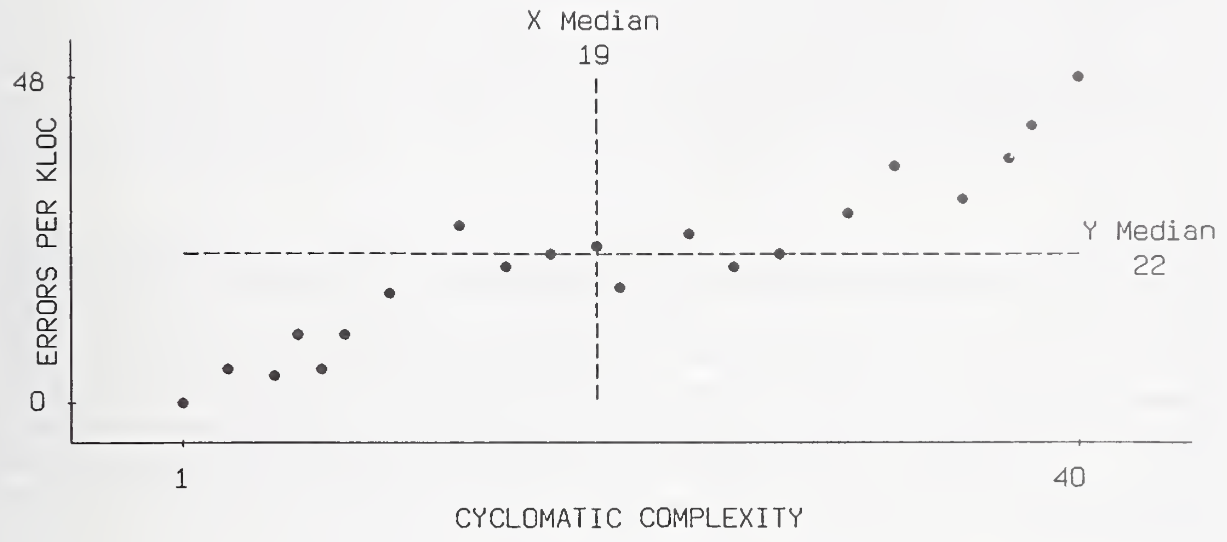

Figure 5-5. Scatter Diagram With Hypothetical Data - Complexity vs. Defect Density.

5.4.6. Method of Least Squares (Regression Technique)

This technique can be used in conjunction with scatter diagrams to obtain a more precise relationship between variables. It is used to determine the equation of the regression line, i.e., the line that "best fits" the data points. With this equation, one can approximate values of one variable when given values of the other. The equation of the line is $Y = a + bX$, where a and b are constants which minimize S, the sum of squares of the deviations of all data points from the regression line. For any sample value x_i of X, the expected Y value is $a + bx_i$. This section is based on [OPMC], [CAPRIO], and [SMITH].

Implementation

1. Collect n data values for each of the 2 variables, X and Y, denoted by $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$.
2. Minimize $S = \Sigma (y_i - a - bx_i)^2$ by first taking the partial derivative of S with respect to a and then with respect to b, setting these derivatives to zero, and then solving for a and b.

3. The results obtained from steps should be the following,[8] where $X_B = \Sigma x_i/n$ and $Y_B = \Sigma y_i/n$:

$$b = \frac{\Sigma (X_i - X_B)(Y_i - Y_B)}{\Sigma (X_i - X_B)^2} \qquad a = Y_B - bX_B$$

Interpretation

The constant *a* represents the intercept of the regression line, i.e., the value of Y when X is 0, and
b represents the slope of the regression line. The idea of this technique is to minimize S, so that all data points will be as close to the regression line as possible. The reason for taking the squares of the deviations, rather than simply the deviations, is so that positive and negative deviations will not cancel each other when they are summed. It would also be possible to sum the absolute values of the deviations, but absolute values are generally harder to work with than squares.

Application Examples

See Application Examples for Scatter Diagrams.

5.5. Software Reliability Estimation Models

"Reliability" is used in a general sense to express a degree of confidence that a part or system will successfully function in a certain environment during a specified time period [JURAN]. Software reliability estimation models can predict the future behavior of a software product, based on its past behavior, usually in terms of failure rates. Since 1972, more than 40 software reliability estimation models have been developed, with each based on a certain set of assumptions characterizing the environment generating the data. However, in spite of much research effort, there is no universally applicable software reliability estimation model which can be trusted to give accurate predictions of reliability in all circumstances [BROCKLEHURST].

It is usually possible to obtain accurate reliability predictions for software, and to have confidence in their accuracy, if appropriate data is used [ROOK]. Also, the use of reliability estimation models is still under active research, so improvements to model capability are likely. Recent work by Littlewood (1989), for example, involves the use of techniques for improving the accuracy of predictions by learning from the analysis of past errors [ROOK], and recalibration [BROCKLEHURST].

[8] Another form of the equation for b, which is often easier to compute is

$$b = \frac{\Sigma X_i Y_i - nX_B Y_B}{\Sigma X_i^2 - n(X_B)^2}$$

Some problems have been encountered by those who have tried to apply reliability estimation models in practice. The algorithms used to estimate the model parameters may fail to converge. When they do, the estimates can vary widely as more data is entered [DACS]. There is also the difficulty of choosing which reliability model to use, especially since one can not know a priori which of the many models is most suitable in a particular context [BROCKLEHURST]. In general, the use of these models is only suitable for situations in which fairly modest reliability levels are required [ROOK].

There are three general classes of software reliability estimation models: nonhomogeneous Poisson process (NHPP) models, exponential renewal NHPP models, and Bayesian models. Some of the more common reliability estimation models are described below [DUNN], [LYU].

- Jelinski-Moranda (JM). One of the earliest models, it assumes the debugging process is purely deterministic, that is, that each defect in the program is equally likely to produce failure (but at random times), and that each fix is perfect, i.e., introduces no new defects. It also assumes that the failure rate is proportional to the number of remaining defects and remains constant between failures. This model tends to be too optimistic and to underestimate the number of remaining faults; this effect has been observed in several actual data sets.

- Goel-Okumoto (GO). This model is similar to JM, except it assumes the failure rate (number of failure occurrences per unit of time) improves continuously in time.

- Yamada Delayed S-Shape. This model is similar to GO, except it accounts for the learning period that testers go through as they become familiar with the software at the start of testing.

- Musa-Okumoto (MO). This NHPP model is similar to GO, except it assumes that later fixes have a smaller effect on a program's reliability than earlier ones. Failures are assumed to be independent of each other.

- Geometric. This model is a variation of JM, which does not assume a fixed, finite number of program errors, nor does it assume that errors are equally likely to occur.

- Schneidewind. Similar to JM, this model assumes that as testing proceeds with time, the error detection process changes, and that recent error counts are usually more useful than earlier counts in predicting future counts.

- Bayesian Jelinski-Moranda (BJM) Model. This model is similar to JM, except that it uses a Bayesian inference scheme, rather than maximum likelihood. Although BJM does not drastically underestimate the number of remaining errors, it does not offer significant improvement over JM. Actual reliability predictions of the two models are usually very close.

- Littlewood. This model attempts to answer the criticisms of JM and BJM by assuming that different faults have different sizes, i.e., they contribute unequally to the unreliability of the software. This assumption represents the uncertainty about the effect of a fix.

- Littlewood-Verrall (LV). This model takes into account the uncertainty of fault size and efficacy of a fix (i.e., a fix is of uncertain magnitude and may make a program less reliable), by letting the size of the improvement in the failure rate at a fix vary randomly.

- Brooks and Motley (BM). The BM binomial and Poisson models attempt to consider that not all of a program is tested equally during a testing period and that only some portions of the program may be available for testing during its development.

- Duane. This model assumes that the failure rate changes continuously in time, i.e., it follows a nonhomogeneous Poisson process. The cumulative failure rate when plotted against the total testing time on a ln-ln graph follows a straight line. The two parameters for the equation of the line can be derived using the method of least squares.

Implementation

The following is a generic procedure for estimating software reliability [AIAA]. It can be tailored to a specific project or lifecycle phase; thus some steps may not be used in some applications.

1. Identify the application. The description of the application should include, at a minimum, the identification of the application, the characteristics of the application domain that may affect reliability, and details of the intended operation of the application system.

2. Specify the requirement. The reliability requirement should be specific enough to serve as a goal (e.g., failure rate of 10^{-9} per hour).

3. Allocate the requirement. The reliability requirement may be distributed over several components, which should be identified.

4. Define failure. A specific failure definition is usually agreed upon by testers, developers, and users prior to the beginning of testing. The definition should be consistent over the life of the project. Classification of failures (e.g., by severity) is continuously negotiated.

5. Characterize the operational environment. The operational environment should be described in terms of the system configuration (arrangement of the system's components), system evolution and system operational profile (how system will be used).

6. Select tests. The test team selects the most appropriate tests for exposing faults. Two approaches to testing can be taken: testing duplicates actual operational environments as

closely as possible; or testing is conducted under more severe conditions than expected in normal operational environments, so that failures can occur in less time.

7. Select the models. The user should compare the models prior to selection based on the following criteria: predictive validity, ease of parameter measurement, quality of the model's assumptions, capability, applicability, simplicity, insensitivity to noise, and sensitivity to parameter variations.

8. Collect data. See section 5.2.

9. Determine the parameters. There are three common methods of estimating the parameters from the data: method of moments, least squares, and maximum likelihood. Each of these methods has useful attributes, but maximum likelihood estimation is the most commonly used approach. However, for some models, the maximum likelihood method does not yield equations for the parameters in closed form, so instead numerical methods (e.g., Newton's method) must be used [ROME]. As stated previously, some datasets may cause the numerical methods not to converge. There exist automated software reliability engineering tools, which are capable of performing parameter estimation.

10. Validate the model. The model should be continuously checked to verify that it fits the data, by using a predictive validity criteria or a traditional statistical goodness-of-fit test (e.g., Chi-square).

11. Perform analysis. The results of software reliability estimation may be used for several purposes, including, but not limited to, estimating current reliability, forecasting achievement of a reliability goal, establishing conformance with acceptance criteria, managing entry of new software features or new technology into an existing system, or supporting safety certification.

Interpretation

A disadvantage of these models is that they rely on testing and hence are used rather late in the development life cycle. The models are usually time based, that is, the probability is based on time to failure. Research is needed to identify how to use more valuable parameters with these models. See [ROOK]

Application Examples

Applicability of the models should be examined through various sizes, structures, functions and application domains. An advantage of a reliability model is its usability in different development and operational environments, and in different lifecycle phases. Software reliability models should be used when dealing with the following situations:

- Evolving software (i.e., software that is incrementally integrated during testing)
- Classification of failure severity
- Incomplete failure data
- Hardware execution rate differences
- Multiple installations of the same software
- Project environments departing from model assumptions

6. SUMMARY

Error analysis for software consists of many activities to assure the quality of delivered software. The activities include error detection, analysis, resolution on an individual level and also on a collective level. In the latter case, the collective data may be used to locate common errors within a product, to identify areas for improvement in software development and maintenance, and to identify areas for improvement in error analysis.

Many activities of error analysis may be conducted during the early phases of the software development lifecycle to prevent error propagation and to reduce costs of fixing the errors at a later time in the lifecycle. Finding the root cause of an error discovered in system test may require analysis of code, design specifications, software requirements documentation, and perhaps analysis and test documentation. Correction of the error results in additional verification and testing activities through the lifecycle products. The time spent initially in preparing correctly stated software requirements will pay off in reduced time needed for rework in the later phases.

The error detection techniques described in this report are a representative sampling of the most widely-used error detection techniques and those most frequently referenced in standards, guidelines and technical literature. The report also describes some techniques which represent new approaches and are not yet widespread. The techniques include those that examine software products without executing the software, those that execute (or simulate the execution of) the software, and those that are based on mathematical analysis and proof of correctness techniques. Evidence of the effectiveness of any of these techniques may be hard to find. Journal articles report success of some techniques, but most often anecdotal evidence is provided through conference presentations and discussions among colleagues.

With many techniques to choose from, appropriate selection for a specific project depends on the characteristics of the project, such as the types of problems most likely to occur. Other selection criteria, which are outside the scope of this report, include parameters like development processes, skill of the technical staff, project resources, and quality requirements of the project.

The study of standards for high integrity software reported in [NUREG, NIST204] indicated that these standards are beginning to require techniques of all kinds with some guidelines attempting to base the requirement on the quality requirements and problem types of the software project. An examination of approximately 50 standards, draft standards, and guidelines indicates that these documents vary widely in their recommendations for error analysis.

Error detection is only one activity of error analysis. Information about the detected error must be reported and delivered to any persons responsible for correcting the error, managing the project, analyzing the data for process improvement and identifying similar problems in the product. Individual problem reports may be collected and analyzed using statistical process control techniques, to determine and monitor the efficiency and adequacy of the development process. Findings which result from using SPC techniques should be used as feedback to improve the development process for the current, as well as future, products. Data on single errors is also used in estimating software reliability and in predicting the number of errors (at later stages in the lifecycle).

Vendors should use error analysis to collect error data and to build corporate memory databases for use across projects. They may use the error information to identify appropriate techniques for similar projects and to better understand how to produce quality software systems.

Regulatory agencies should also consider establishing and maintaining a database of error analysis data for software systems. Willingness of vendors to participate must include mechanisms to assure confidentiality of proprietary information and that vendor data will not be used in a recriminatory sense. The database must contain both developmental and operational error data for effective use. Vendors, auditors, regulators, and the software engineering community may all benefit from use of error databases.

Auditors may use the information in the database to identify the most error-prone features of specific high integrity systems and may ensure that their audits examine these features carefully. The auditors may use the data to identify acceptance limits on different aspects of the high integrity software safety system.

Regulators may use the information from a database in several ways. First, over time, it may become apparent that some error analysis techniques are more effective than others with respect to a given type of problem. It may also become apparent that problems in these areas occur most often with certain development practices or occur less frequently with other development practices. This knowledge may influence recommendations in regulatory documents.

Finally, careful analysis of the information in the database may enable the software engineering community in general to identify research needs for software development practices to prevent specific problems from occurring and error analysis techniques to locate problems as soon as possible in the software lifecycle.

7. REFERENCES

[AIAA]

R-013-1992, "Recommended Practice for Software Reliability," American Institute of Aeronautics and Astronautics, Space-Based Observation Systems Committee on Standards, Software Reliability Working Group, 1992, c/o Jim French, AIAA Headquarters, 370 L'Enfant Promenade, SW, Washington, DC 20024-2518W.

[AIRFORCE]

AFSCP 800-14, Air Force Systems Command, Software Quality Indicators, "Management Quality Insight," Department of the Air Force, January 20, 1987.

[ALBRECHT]

Albrecht, Allan J. and John E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November, 1983.

[ANS104]

ANSI/ANS-10.4-1987, "Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry," American Nuclear Society, May 13, 1987.

[ASMENQA2]

ASME NQA-2a-1990 Part 2.7, "Quality Assurance Requirements for Nuclear Facility Applications," The American Society of Mechanical Engineers, November 1990.

[BASILI]

Basili, V.R. and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, IEEE Computer Society, Volume SE-13, Number 12, December 1987, pp. 1278-1296.

[BEIZER]

Beizer, Boris, Software Testing Techniques, Van Nostrand Reinhold Company, New York, 1983.

[BROCKLEHURST]

Brocklehurst, S., P. Y. Chan, Bev Littlewood, and John Snell, "Recalibrating Software Reliability Models," IEEE Transactions on Software Engineering, Vol. 16, No. 4, 1990.

[CAPRIO]

Caprio, William H., "The Tools for Quality," Total Quality Management Conference, Ft. Belvoir, Virginia, July 13-15, 1992.

[CLARK]
Clark, Peter, and Bard S. Crawford, Evaluation and Validation (E&V) Reference Manual, TASC No. TR-5234-3, Version 3.0, February 14, 1991.

[DACS]
"Software Reliability Models," DACS Newsletter, Data & Analysis Center for Software, Volume X, Number 2, Summer, 1992.

[DEMILLO]
DeMillo, Richard A. et al. Software Testing and Evaluation, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[DEMMY]
Demmy, W. Steven and Arthur B. Petrini, "Statistical Process Control in Software Quality Assurance," Proceedings of the 1989 National Aerospace and Electronics Conference, NAECON, May 22-26, 1989, Dayton, OH, IEEE, Inc., Piscataway, NJ, p. 1585-1590.

[DUNN]
Dunn, Robert. Software Defect Removal, McGraw-Hill, Inc., 1984.

[EWICS3]
Bishop, P. G. (ed.), Dependability of Critical Computer Systems 3 - Techniques Directory, The European Workshop on Industrial Computer Systems Technical Committee 7 (EWICS TC7), Elsevier Science Publishers Ltd, 1990.

[FAGAN]
Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Volume 15, Number 3, 1976.

[FLORAC]
CMU/SEI-92-TR-ZZZ, "Software Quality Measurement: A Framework for Counting Problems, Failures, and Faults," William Florac, The Quality Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Draft, 1992.

[FREEDMAN]
Freedman, David, Robert Pisani, and Roger Purves, "Statistics," W.W. Norton & Company, Inc., New York, 1978.

[GRADY]
Grady, Robert B. and Caswell, Deborah, Software Metrics: Establishing a Company-Wide Program, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[GREEN]
Green, Scott E. and Rose Pajerski, "Cleanroom Process Evolution in the SEL," Proceedings of the Sixteenth Annual Software Engineering Workshop, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, MD 20771, December 1991.

[IEC65A94]
IEC 65A(Secretariat)94, "Draft British Standard 0000: Software for Computers in the Application of Industrial Safety-Related Systems," WG9, December 6, 1989.

[IEC65A122]
IEC 65A(Secretariat)122, "Software for Computers in the Application of Industrial Safety-Related Systems," WG9, Version 1.0, September 26, 1991.

[IEEEGLOSS]
ANSI/IEEE Std 610.12, "IEEE Standard Glossary of Software Engineering Terminology," The Institute of Electrical and Electronics Engineers, February, 1991.

[IEEEP1044]
IEEE P1044, "Draft Standard of: A Standard Classification for Software Errors, Faults, and Failures," The Institute of Electrical and Electronics Engineers, August 1991.

[IEEE982.2]
ANSI/IEEE Std 982.2-1988, "Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June, 1989.

[IEEE1012]
ANSI/IEEE Std 1012-1986, "IEEE Standard for Software Verification and Validation Plans," The Institute of Electrical and Electronics Engineers, Inc., November 14, 1986.

[JURAN]
Juran, J. M. (ed.), Juran's Quality Control Handbook, 4th ed., McGraw-Hill, Inc., New York, 1988.

[KELLY]
Kelly, John C., Joseph S. Sherif, and Jonathan Hops, "An Analysis of Defect Densities Found During Software Inspections," Proceedings of the Fifteenth Annual Software Engineering Workshop, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, MD 20771, November, 1990.

[KITCHENHAM]
Kitchenham, B. A. and B. Littlewood, Measurement for Software Control and Assurance, Elsevier Science Publishers Ltd, London and New York, 1989.

[LYLE]
Lyle, Jim, "Program Slicing," to be published in Encyclopedia of Software Engineering, John Wiley Publishing Co., New York, New York.

[LYU]
Lyu, Michael and Allen Nikora, "Applying Reliability Models More Effectively," IEEE Software, Vol. 9., No. 4, July, 1992.

[MAKOWSKY]
Makowsky, Lawrence C., Technical Report, USA-BRDEC-TR//2516, "A Guide to Independent Verification and Validation of Computer Software," United States Army, Belvoir Research, Development and Engineering Center, June 1992.

[MILLER]
Miller, Keith W., et al, "Estimating the Probability of Failure When Testing Reveals No Failures," IEEE Transactions on Software Engineering, Vol.18, No.1, January 1992.

[MILLS]
Mills, H. D., M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," IEEE Software, September, 1987, pp. 19-25.

[MOD55]
Interim Defence Standard 00-55, "The Procurement of Safety Critical Software in Defence Equipment," Parts 1 and 2, Ministry of Defence, UK, April 5, 1991.

[MYERS]
Myers, Glenford J., The Art of Software Testing, John Wiley & Sons, New York, 1979.

[NBS93]
Powell, Patricia B., NBS Special Publication 500-93, "Software Validation, Verification, and Testing Technique and Tool Reference Guide," U.S. Department of Commerce/National Bureau of Standards (U.S.), September 1982.

[NIST187]
NIST SPEC PUB 500-187, "Application Portability Profile (APP) The U.S. Government's Open System Environment Profile OSE/1 Version 1.0," U.S. Department of Commerce/National Institute of Standards and Technology, April 1991.

[NIST204]
Wallace, D.R., L.M. Ippolito, D.R. Kuhn, NIST SPEC PUB 500-204, "High Integrity Software Standards and Guidelines," U.S. Department of Commerce/National Institute of Standards and Technology, September, 1992.

[NISTGCR]
Craigen, Dan, Susan Gerhart, Ted Ralston, NISTGCR 93/626, "An International Survey of Industrial Applications of Formal Methods," Volumes 1 and 2, U.S. Department of Commerce/National Institute of Standards and Technology, March, 1993.

[NISTIR]
Wallace, D.R., W.W. Peng, L.M. Ippolito, NISTIR 4909,"Software Quality Assurance: Documentation and Reviews," U.S. Department of Commerce/National Institute of Standards and Technology, 1992.

[NUREG]
Wallace, D.R., L.M. Ippolito, D.R. Kuhn, NUREG/CR-5930, "High Integrity Software Standards and Guidelines," U.S. Nuclear Regulatory Commission, September, 1992.

[OPMC]
The Organizational Process Management Cycle Programmed Workbook, Interaction Research Institute, Inc., Fairfax, Virginia.

[PARK]
Park, Robert, CMU/SEI-92-TR-20, ESC-TR-92-20, "Software Size Measurement: A Framework for Counting Source Statements," Software Engineering Institute, Carnegie Mellon University, September, 1992.

[PEYTON]
Peyton and Hess, "Software Sneak Analysis," IEEE Seventh Annual Conference of the Engineering in Medicine and Biology Society, The Institute of Electrical and Electronics Engineers, 1985.

[PUTNAM]
Putnam, Lawrence H. and Ware Myers, Measures for Excellence, Reliable Software On Time, Within Budget, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[RIFKIN]
Rifkin, Stan and Charles Cox, "Measurement in Practice," Technical Report, CMU/SEI-91-TR-16, ESD-TR-91-16, Carnegie Mellon University, 1991.

[ROME]
Iuorno, Rocco and Robert Vienneau, "Software Measurement Models," Draft, Prepared for Rome Air Development Center, Griffiss Air Force Base, NY, July 1987.

[ROOK]
Rook, Paul, Software Reliability Handbook, Elsevier Science Publishers Ltd, Crown House, London and New York, 1990.

[SCHULMEYER]
Schulmeyer, G. Gordon. Zero Defect Software, McGraw-Hill, Inc., 1990.

[SHOOMAN]
Shooman, Martin L., "A Class of Exponential Software Reliability Models," Workshop on Software Reliability, IEEE Computer Society Technical Committee on Software Reliability Engineering, Washington, DC, April 13, 1990.

[SQE]
"Software Measurement," Seminar Notebook, Version 1.2, Software Quality Engineering, 1991.

[SMITH]
Smith, Gary, Statistical Reasoning, Allyn and Bacon, Boston, MA, 1991.

[STSC]
MAC010/STSC, "Software Management Guide," Software Technology Center, Hill AFB, UT 84-56, October, 1990.

[ROME]
Proceedings of the 3rd Annual Software Quality Workshop; Alexandria Bay, New York: August 11-15, 1991, Rome Laboratory, Griffiss AFB, New York.

[VOAS]
Voas, Jeffrey M. and Keith W. Miller," A Model for Improving the Testing of Reusable Software Components," 10th Pacific Northwest Software Quality Conference, October 19-21, 1992, Portland, Oregon.

[WALLACE]
Wallace, Dolores R. "Software Verification and Validation," to be published in Encyclopedia of Software Engineering, John Wiley Publishing Co., New York, NY.

[WALLACEFUJII]
Wallace, Dolores R. and Roger U. Fujii, "Verification and Validation: Techniques to Assure Reliability," IEEE Software, May 1989.

[ZAGE]
Zage, Wayne M. "Code Metrics and Design Metrics; An ACM Professional Development Seminar," November 19, 1991.

APPENDIX A. ERROR DETECTION TECHNIQUES

A.1. Algorithm Analysis

Description:
The purpose is to determine the correctness of an algorithm with respect to its intended use, to determine its operational characteristics, or to understand it more fully in order to modify, simplify, or improve. The analysis involves rederiving equations or evaluating the suitability of specific numerical techniques. Algorithms are analyzed for correctness, efficiency (in terms of time and space needed), simplicity, optimality, and accuracy. Algorithm analysis also examines truncation and round-off effects, numerical precision of word storage and variables (e.g., single- vs. extended-precision arithmetic), and data typing influences.

Advantages:
- Effective and useful in general

Disadvantages:
- A particular analysis depends on the particular model of computation (e.g, Turing machine, random access machine). If the assumptions of the model are inappropriate, then the analysis will be inaccurate.

Type of Errors Detected:
- Incorrect, inappropriate, or unstable algorithms
- Program does not terminate
- Inability to operate on the full range of data - e.g., trigonometric routine only works in the first quadrant
- Incorrect analysis of computational error (effect of round-off and truncation errors)
- Incompatible data representations - e.g., input in lbs., program processes kilograms
- Incompatibility with hardware or software resources

References: [IEEE1012], [DUNN], [WALLACE], [NBS93]

A.2. Back-to-Back Testing

Description:
This technique is used to detect test failures by comparing the output of two or more programs implemented to the same specification. The same input data is applied to two or more program versions and their outputs are compared to detect anomalies. Any test data selection strategy can be used for this type of testing, although random testing is well suited to this approach. Also known as comparison testing.

Advantages:
- Permits a large number of tests to be made with little effort

- Rapid fault detection

Disadvantages:
- Requires the construction of at least one secondary program, although this may be available as part of the overall development
- Discrepancies must be analyzed manually to determine which program is at fault (it is not sufficient to assume that majority is always correct)

Types of Errors Detected:
- Does not detect specific errors, only anomalies or discrepancies between programs

References: [EWICS3]

A.3. Boundary Value Analysis

Description:
The purpose is to detect and remove errors occurring at parameter limits or boundaries. The input domain of the program is divided into a number of input classes. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries of the input domain of the specification coincide with those in the program. The value zero, whether used directly or indirectly, should be used with special attention (e.g., division by zero, null matrix, zero table entry). Usually, boundary values of the input produce boundary values for the output. Test cases should also be designed to force the output to its extreme values. If possible, a test case which causes output to exceed the specification boundary values should be specified. If output is a sequence of data, special attention should be given to the first and last elements and to lists containing zero, one, and two elements.

Advantages:
- Verifies that program will behave correctly for any permissible input or output

Disadvantages:
- No significant disadvantages in itself but for programs with many types of input cannot test all combinations of input and therefore cannot identify problems resulting from unexpected relationships between input types

Types of Errors Detected
- Algorithm errors
- Array size
- Specification errors

References: [MYERS]

A.4. Control Flow Analysis/Diagrams

Description:
This technique is most applicable to real time and data driven systems. Logic and data requirements are transformed from text into graphic flows, which are easier to analyze. Examples of control flow diagrams include PERT, state transition, and transaction diagrams. For large projects, control flow analysis using control flow diagrams that show the hierarchy of main routines and subfunctions are useful to understand the flow of program control. Purpose is to detect poor and potentially incorrect program structures. The program is represented by a directed graph, which is analyzed for the errors below.

Advantages:
- Simple to apply
- Readily automated

Disadvantages:
- Results require some interpretation. Identified anomalies may not be faults.
- Sometimes difficult to deal with "aliasing" where different variables are associated with the same locations in memory

Types of Errors Detected:
- Inaccessible/unreachable code
- Knotted code - If code is well-structured the directed graph can be reduced to a single node. If code is poorly structured, it can only be reduced to a "knot" composed of several nodes.

References: [IEEE1012], [EWICS3]

A.5. Database Analysis

Description:
Database analysis is performed on programs with significant data storage to ensure that common data and variable regions are used consistently between all call routines; data integrity is enforced and no data or variable can be accidentally overwritten by overflowing data tables; data access through indirect access is checked; and data typing and use are consistent throughout all program elements. Useful for programs that store program logic in data parameters. The purpose is to ensure that the database structure and access methods are compatible with the logical design. Diagrams are useful for understanding user privileges.

Advantages:
- Supports interface analysis

Disadvantages:
- May require manual examination of diagrams for access problems

Types of Errors Detected:
- Inconsistent use of data types
- Incorrect access protections

References: [IEEE1012]

A.6. Data Flow Analysis

Description:
The purpose is to detect poor and potentially incorrect program structures. Data flow analysis combines the information obtained from the control flow analysis with information about which variables are read or written in different portions of code. May also be used in the design and implementation phases.

Advantages:
- Readily automated
- Easy to apply

Disadvantages:
- Requires some interpretation
- Some anomalies may not be faults

Types of Errors Detected:
- Undefined input/output data or format
- Incorrect data flow
- Variables that are read before they are written (likely to be an error, and is bad programming practice)
- Omitted code - indicated by variables that are written more than once without being read
- Redundant code - indicated by variables that are written but never read

References: [EWICS3], [IEEE1012]

A.7. Data Flow Diagrams

Description:
Data flow diagrams are used to describe the data flow through a program in a diagrammatic form. They show how data input is transformed to output, with each stage representing a distinct transformation. The diagrams use three types of components:

1. Annotated bubbles - bubbles represent transformation centers and the annotation specifies the transformation
2. Annotated arrows - arrows represent the data flow in and out of the transformation centers, annotations specify what the data is
3. Operators (AND, XOR) - used to link the annotated arrows

Data flow diagrams only describe data, and should not include control or sequencing information. Each bubble can be considered a black box which, as soon as its inputs are available, transforms them to outputs. Each should represent a distinct transformation, whose output is somehow different from its input. There are no rules regarding the overall structure of the diagram.

Advantages:
- They show transformations without making assumptions about how the transformations are implemented.

Disadvantages:
- Inability to provide information about the transformation process

Type of Errors Detected:
- Incorrect data input/output
- Inconsistencies in data usage

References: [IEC65A94]

A.8. Decision Tables (Truth Tables)

Description:
The purpose is to provide a clear and coherent analysis of complex logical combinations and relationships. This method uses two-dimensional tables to concisely describe logical relationships between boolean program variables.

Advantages:
- Their conciseness and tabular nature enable the analysis of complex logical combinations expressed in code.
- Potentially executable if used as specifications

Disadvantages:
- Tedious effort required

Types of Errors Detected:
- Logic

References: [IEC65A122]

A.9. Desk Checking (Code Reading)

Description:
Code is read by an expert, other than the author of the code, who performs any of the following: looking over the code for obvious defects, checking for correct procedure interfaces, reading the comments to develop a sense of what the code does and then comparing it to its external specifications, comparing comments to design documentation, comparing comments to design documentation, stepping through with input conditions contrived to "exercise" all paths including those not directly related to the external specifications, checking for compliance with programming standards and conventions, any combination of the above.

Advantages:
- Inexpensive
- Capable of catching 30% of all errors, if performed meticulously
- Can be more effective than functional testing or structural testing (NASA Goddard Space Flight Center experiment, see [BASILI])

Disadvantages:
- Requires enormous amount of discipline
- Few people are able to use this technique effectively
- Usually less effective than walkthroughs or inspections

Types of errors detected:
LOGIC AND CONTROL
- unreachable code
- improper nesting of loops and branches
- inverted predicates
- incomplete predicates
- improper sequencing of processes
- infinite loops
- instruction modification
- failure to save or restore registers
- unauthorized recursion
- missing labels or code
- unreferenced labels

COMPUTATIONAL
- missing validity tests
- incorrect access of array components
- mismatched parameter lists
- initialization faults
- anachronistic data
- undefined variables
- undeclared variables

- misuse of variables (locally and globally)
- data fields unconstrained by natural or defined data boundaries
- inefficient data transport

OTHER
- calls to subprograms that do not exist
- improper program linkages
- input-output faults
- prodigal programming
- failure to implement the design

References: [WALLACE], [DUNN], [BEIZER], [BASILI]

A.10. Error Seeding

Description:
The purpose of this technique is to determine whether a set of test cases is adequate. Some known error types are inserted into the program, and the program is executed with the test cases under test conditions. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is approximate equal to the ratio of found real errors to total number of errors, or

$$\frac{\text{Number of seeded errors found}}{\text{Total number of seeded errors}} = \frac{\text{Number of real errors found}}{\text{Total number of real errors}}$$

In the equation, one can solve for the total number of real errors, since the values of the other three are known. Then, one can estimate the number of errors remaining by subtracting the number of real errors found from the total number of real errors. The remaining test effort can then be estimated. If all the seeded errors are found, this indicates that either the test case set is adequate, or that the seeded errors were too easy to find.

Advantages:
- Provides indication that test cases are structured adequately to locate errors.

Disadvantages:
- For this method to be valid and useful, the error types and the seeding positions must reflect the statistical distribution of real errors.

Types of Errors Detected:
- Does not detect errors, but determines adequacy of test set

References: [IEC65A122]

A.11. Finite State Machines

Description:
The purpose is to define or implement the control structure of a system. Many systems can be defined in terms of their states, inputs, and actions. For example, a system is in state S_1, receives an input I, then carries out action A, and moves to state S_2. By defining a system's actions for every input in every state we can completely define a system. The resulting model of the system is a finite state machine (FSM). It is often drawn as a state transition diagram, which shows how the system moves from one state to another, or as a matrix in which the dimensions are state and input. Each matrix entry is identified by a state and input, and it specifies the action and new state resulting from receipt of the input in the given state.

Advantages:
- Allows important properties to be checked mechanically and reliably.
- Simple to work with

Disadvantages:
- None that are major

Types of Errors Detected:
- Incomplete requirements specification - check that there is an action and new state for every input in every state
- Inconsistent requirements - check that only one state change is defined for each state and input pair

References: [EWICS3]

A.12. Formal Methods (Formal Verification, Proof of Correctness, Formal Proof of Program)

Description:
The purpose is to check whether software fulfills its intended function. Involves the use of theoretical and mathematical models to prove the correctness of a program without executing it. The requirements should be written in a formal specification language (e.g., VDM, Z) so that these requirements can then be verified using a proof of correctness. Using this method, the program is represented by a theorem and is proved with first-order predicate calculus. A number of assertions are stated at various locations in the program, and are used as pre and post conditions to various paths in the program. The proof consists of showing that the program transfers the preconditions into the postconditions according to a set of logical rules, and that the program terminates.

Advantages:
- Allows for rigorous statement concerning correctness

- Possibly the only way of showing the correctness of general WHILE loops

Disadvantages:
- Time consuming to do manually
- Requires enormous amount of intellectual effort
- Must be checked mechanically for human errors
- Difficult to apply to large software systems
- If formal specifications are not used in the design, then formal verification (proof of correctness after implementation tends to be extremely difficult
- Only applicable to sequential programs, not concurrent program interactions

References: [IEC65A122], [ROOK]

A.13. Information Flow Analysis

Description:
An extension of data flow analysis, in which the actual data flows (both within and between procedures) are compared with the design intent. Normally implemented with an automated tool where the intended data flows are defined using a structured comment that can be read by the tool.

Advantages:
- Simple to apply
- Readily automated

Disadvantages:
- Results require some interpretation
- Some anomalies may not be faults

Types of Errors Detected:
- Undefined input / output data or format
- Incorrect flow of information

References: [EWICS3]

A.14. (Fagan) Inspections

Description:
An inspection is an evaluation technique in which software requirements, design, code, or other products are examined by a person or group other than the author to detect faults, violations of development standards, and other problems. An inspection begins with the distribution of the item to be inspected (e.g., a specification, some code, test data). Each participant is required to analyze the item on his own. During the inspection, which is the meeting of all the participants,

the item is jointly analyzed to find as many errors as possible. All errors found are recorded, but no attempt is made to correct the errors at that time. However, at some point in the future, it must be verified that the errors found have actually been corrected. Inspections may also be performed in the design and implementation phases.

Advantages:

- Provides comprehensive statistics on classes of errors
- Studies have shown that inspections are an effective method of increasing product quality (e.g., reliability, usability, maintainability)
- Effective for projects of all sizes
- Qualitative benefits: less complex programs, subprograms written in a consistent style, highly visible system development, more reliable estimating and scheduling, increased user satisfaction, improved documentation, less dependence on key personnel for critical skills

Disadvantages:

- Inspectors must be independent of programmers
- Programmers may feel inspection is a personal attack on their work
- Time consuming, involving several staff (2 or 3 pages of not-difficult code may take 3 h to inspect)

Type of Errors Detected:

- Weak modularity
- Failure to handle exceptions
- Inexpansible control structure
- Nonexisting or inadequate error traps
- Incomplete requirements
- Infeasible requirements
- Conflicting requirements
- Incorrect specification of resources

References: [DUNN], [FAGAN], [IEC65A94], [MYERS], [NBS93]

A.15. Interface Analysis

Description:
This technique is used to demonstrate that the interfaces of subprograms do not contain any errors or any errors that lead to failures in a particular application of the software or to detect all errors that may be relevant. Interface analysis is especially important if interfaces do not contain assertions that detect incorrect parameter values. It is also important after new configurations of pre-existing subprograms have been generated. The types of interfaces that are analyzed include external, internal, hardware/hardware, software/software, software/hardware, and software/database. Interface analysis may include the following:

- Analysis of all interface variables at their extreme positions
- Analysis of interface variables individually at their extreme values with other interface variables at normal values
- Analysis of all values of the domain of each interface variable with other interface variables at normal values

Advantages:
- Can locate problems that would prevent system from functioning due to improper interfaces
- Especially useful for software requirements verification and design verification
- When used with prototyping or simulation, can find many critical errors that would be costly to correct in the delivered system
- Software design tools exist for analysis of interfaces, during design phase

Disadvantages:
- Manual effort may be time-consuming

Types of Errors detected:
- Input / output description errors (e.g., values of input variables altered)
- Actual and formal parameters mismatch (precision, type, units, and number)
- Incorrect functions used or incorrect subroutine called
- Inconsistency of attributes of global variables (common, etc.)
- Incorrect assumptions about static and dynamic storage of values (i.e., whether local variables are saved between subroutine calls)
- Inconsistencies between subroutine usage list and called subroutine

References: [WALLACE], [MAKOWSKY]

A.16. Interface Testing

Description:
Similar to interface analysis, except test cases are built with data that tests all interfaces. Interface testing may include the following:

- Testing all interface variables at their extreme positions
- Testing interface variables individually at their extreme values with other interface variables at normal values
- Testing all values of the domain of each interface variable with other interface variables at normal values
- Testing all values of all variables in combination (may be feasible only for small interfaces).

Advantages:

- Locates errors that may prevent system from operating at all or locates errors in timing of interface responses (e.g., slow system response to users was a factor in the failure of the THERAC medical system)

Disadvantages:

- Without automation of design, or code modules, manual searching for interface parameters in all design or code modules can be time consuming

Types of Errors Detected:

- Input / output description errors
- Inconsistent interface parameters

References: [IEC65A122]

A.17. Mutation Analysis

Description:
The purpose is to determine the thoroughness with which a program has been tested, and in the process, detect errors. This procedure involves producing a large set of versions or "mutations" of the original program, each derived by altering a single element of the program (e.g., changing an operator, variable, or constant). Each mutant is then tested with a given collection of test data sets. Since each mutant is essentially different from the original, the testing should demonstrate that each is in fact different. If each of the outputs produced by the mutants differ from the output produced by the original program and from each other, then the program is considered adequately tested and correct.

Advantages:

- Applicable to any algorithmic solution specification
- Results are good predictors of operational reliability

Disadvantages:

- Likely to require significant amounts of human analyst time and good insight
- Requires good automated tools to be effective
- Reliable only if all possible mutation errors are examined
- Cannot assure the absence of errors which cannot be modeled as mutations

Type of Errors Detected:

- Any errors that can be found by test

References: [ANS104], [NBS93], [DEMILLO]

A.18. Performance Testing

Description:
The purpose is to measure how well the software system executes according to its required response times, cpu usage, and other quantified features in operation. These measurements may be simple to make (e.g., measuring process time relative to volumes of input data) or more complicated (e.g., instrumenting the code to measure time per function execution).

Advantages:
- Useful for checking timing synchronization of functions, memory locations, memory requirements and other performance features.

Disadvantages:
- Caution required in instrumenting code to ensure the instrumentation itself does not interfere with processing of functions or with locations of bytes under examination

Types of Errors Detected:
- Timing, synchronization, and memory allocation

References: [ROOK]

A.19. Prototyping / Animation

Description:
Purpose is to check the feasibility of implementing a system against the given constraints and to communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings. A subset of system functions, constraints, and performance requirements are selected. A prototype is built using high level tools, is evaluated against the customer's criteria, and the system requirements may be modified as a result of this evaluation.

Advantages:
- Better communication with customer
- Early detection of problems
- Check feasibility of new ideas or techniques

Disadvantages:
- Unnecessary and expensive if problem is well understood
- Tools are needed for quick implementation

Types of Errors Detected:
- User related
- Interface related
- Omitted functions

- Undesired functions
- Poorly defined functionality (e.g., specifications that do not cover all expected cases)
- Errors in the specification that lead to inconsistent states, failure conditions, erroneous results
- Contradictions in requirements
- Impossibility, infeasibility of requirements

A.20. Regression Analysis and Testing

Description:
Regression analysis is used to reevaluate requirements and design issues whenever any significant code change is made. It involves retesting of a software product to detect faults made during modification, to verify that modification has not caused unintended side effects, and to verify that the modified software still meets its specified requirements. Any changes made during installation and test are reviewed using regression analysis and test to verify that the basic requirements and design assumptions, which affect other areas of the program, have not been violated.

Advantages:
- Effectiveness depends on the quality of the data used. If tests based on the functional requirements are used to create the test data, technique is highly effective

Disadvantages:
- Expense can appear to be prohibitive, especially for small changes. However, it can often be determined which functions may be affected by a given change, so that the amount of testing can be reduced in these cases

Types of Errors Detected:
- Errors caused by system modifications or corrections

References: [ANS104], [WALLACE], [IEEE1012], [NBS93]

A.21. Requirements Parsing

Description:
This technique involves the examination of individual requirements statements to ensure that each statement is complete, readable, accurate, and consistent with other requirements. The manual technique requires analysis of the attributes of a statement: initiator of action, the action, the object of the action, conditions (e.g., when positive state reached), constraints (limits), source of action, destination (e.g., printer, screen, plotter), mechanism, reason for the action. When the data from a set of requirements is examined collectively, the results of the analysis may aid in identifying trends.

Advantages:
- Promotes clarity, correctness, completeness, testability, and accuracy
- Can help to establish an English base from which to write a formal specification

Disadvantages:
- Very narrow look at each requirement. Could detract from analysis of more global examination of how the requirements fit together.

Types of errors detected:
- Inconsistency in requirements
- Incomplete requirement
- Untestable requirement

A.22. Reviews

Description:
A review is a meeting at which the requirements, design, code, or other products of a software development project are presented to the user, sponsor, or other interested parties for comment and approval, often as a prerequisite for concluding a given phase of the software development process. Usually held at end of a phase, but may be called when problems arise. Often referred to as "Formal Review" versus desktop review of materials.

Advantages:
- Provides opportunity to change course of a project before start of next phase.
- Because scope of review is usually broad, gives opportunity to recognize global problems

Disadvantages:
- If participants do not have materials ahead of time and spend time preparing, review will accomplish little or nothing.
- Attention focus on major issues

References: [ANS104], [IEEE1028]

A.23. Sensitivity Analysis

Description:
Sensitivity analysis is a new method of quantifying ultra-reliable software during the implementation phase. It is based on the fault-failure model of software and attempts to approximate this model. There are three necessary conditions:

1. The fault must be executed.
2. The fault must affect the computational data state directly after the fault location.
3. The affected data state must propagate to an output variable.

Sensitivity analysis is based on the premise that software testability can predict the probability that failure will occur when a fault exists given a particular input distribution. A sensitive location is one in which faults cannot hide during testing. The approach expects reasonably "close to correct" code; results are tied to selected input distribution. The internal states are perturbed to determine sensitivity. Researchers of this technique have developed a tool that performs several analyses on source code to estimate the testability and sensitivity of the code. These analyses require instrumentation of the code and produce a count of the total executions through an operation (execution analysis), an infection rate estimate, and a propagation analysis.

Advantages:

- While software testing provides some quantification of reliability, sensitivity analysis provides quantification of software testing.
- Presents a different approach to software reliability assessment, which usually is based on an exponential distribution of remaining faults over time.
- Promising research that those needing ultra-reliable software should continue to follow.

Disadvantages:

- Intended to operate on code that has already been formally verified
- Still new, so that effectiveness of this technique is not yet known and use is not yet widespread
- Developed and currently promoted by only one company
- Tool is not sold, but rather the service of operating the tool on a client's code is sold

Types of Errors Detected:

- None, but is an aid to error detection techniques

References: [MILLER], [VOAS]

A.24. Simulation

Description:

Simulation is used to test the functions of a software system, together with its interface to the real environment, without modifying the environment in any way. The simulation may be software only or a combination of hardware and software. Simulation is used to evaluate the interactions of large, complex systems with many hardware, user, and other interfaces. A model of the system to be controlled by the actual system under test is created. This model mimics the behavior of the controlled system, and is for testing purposes only. It must provide all inputs of the system under test which will exist when that system is installed; respond to outputs from the system in a way which accurately represents the controlled system; have provision for operator inputs to provide any perturbations with which the system under test is required to cope. When software is being tested, the simulation may be a simulation of the target hardware with its inputs and outputs. In the installation phase, it is used to test operator procedures and to help isolate installation problems.

Advantages:

- Enables simulation of critical aspects of the software that would not be practical to analyze manually.
- Provides means of performing functional tests of the system's behavior in the event of catastrophes which could not otherwise not be tested
- Can provide a means of achieving "long term" test in a short period
- Can provide a means of investigating behavior at critical points by slowing the timescale or single stepping

Disadvantages:

- Difficult to achieve independently of the software being tested
- May require considerable resource both to create the simulation and to operate it
- Dependent on a model of the system which itself could contain safety related flaws

Types of Errors Detected:

- Interface errors
- Logical errors
- Errors in performance

References: [IEEE1012], [WALLACE], [IEC65A122], [IEC65A94]

A.25. Sizing and Timing Analysis

Description:
Sizing/timing analysis is performed during incremental code development by obtaining program sizing and execution timing values to determine if the program will satisfy processor size and performance requirements allocated to the software. Significant deviations between actual and predicted values is a possible indication of problems or the need for additional examination. This technique is most applicable to real-time programs having response time requirements and constrained memory execution space requirements.

Advantages:

- Opportunity to study sequence of operations, especially interfaces between humans and response to user commands
- Indicator of whether integrated system will perform appropriately

Disadvantages:

- Time consuming when conducted by poring over control flow or data flow diagrams
- When timing results are collected by executing code, must be careful that test code does not alter timing as it would be without the test code

Types of Error Detected:

- Unacceptable processor load
- Control structure ignores processing priorities

References: [WALLACE], [DUNN], [IEEE1012]

A.26. Slicing

Description:
Slicing is a program decomposition technique that is based on extracting statements relevant to a specific computation from a program. It produces a smaller program that is a subset of the original program. Without intervening irrelevant statements, it is easier for a programmer interested in a subset of the program's behavior to understand the corresponding slice rather than to deal with the entire program. This technique can be applied to program debugging, testing, parallel program execution and software maintenance. Several variations of this technique have been developed, including program dicing, dynamic slicing and decomposition slicing.

Advantages:
- Readily automated
- Reduces time needed for debugging and testing

Disadvantages:
- Resource consuming tool/ method

Types of Errors Detected:
- Aids in finding root of errors during debugging and testing, by narrowing the focus of investigation

References: [LYLE]

A.27. Software Sneak Circuit Analysis

Description:
This technique is used to detect an unexpected path or logic flow which causes undesired program functions or inhibits desired functions. Sneak circuit paths are latent conditions inadvertently designed or coded into a system, which can cause it to malfunction under certain conditions.

To perform sneak circuit analysis, source code is first converted, usually by computer, into an input language description file into topological network trees. Then the trees are examined to identify which of the six basic topological patterns appear in the trees. Analysis takes place using checklists of questions about the use and relationships between the basic topological components.

Advantages:
- Effective in finding errors not usually detected by desk checking or standard V&V techniques

- Applicable to programs written in any language
- Applicable to hardware, software, and the combined system

Disadvantages:
- Labor intensive
- Likely to be performed late in the development cycle, so changes will be costly

Types of Error Detected:
- Unintended functions/outputs

References: [PEYTON], [EWICS3]

A.28. Stress Testing

Description:
Involves testing the response of the system to extreme conditions (e.g., with an exceptionally high workload over a short span of time) to identify vulnerable points within the software and to show that the system can withstand normal workloads. Examples of testing conditions that can be applied include the following:

- If the size of the database plays an important role, then increase it beyond normal conditions.
- Increase the input changes or demands per time unit beyond normal conditions.
- Tune influential factors to their maximum or minimal speed.
- For the most extreme case, put all influential factors to the boundary conditions at the same time.

Under these test conditions, the time behavior can be evaluated and the influence of load changes observed. The correct dimension of internal buffers or dynamic variables, stacks, etc. can be checked.

Advantages:
- Often the only method to determine that certain kinds of systems will be robust when maximum numbers of users are using the system, at fastest rate possible (e.g., transaction processing) and to identify that contingency actions planned when more than maximum allowable numbers of users attempt to use system, when volume is greater than allowable amount, etc.

Disadvantages:
- Requires large resources

Types of Errors Detected:
- Design errors related to full-service requirements of system and errors in planning defaults when system is over-stressed

References: [MYERS]

A.29. Symbolic Execution

Description:
This is an evaluation technique in which program execution is simulated using symbols rather than actual values for input data, and program output is expressed as logical or mathematical expressions involving these symbols.

Advantages:
- No input data values are needed
- Results can be used in program proving
- Useful for discovering a variety of errors

Disadvantages:
- Result will consist of algebraic expressions which easily get very bulky and difficult to interpret
- Difficult to analyze loops with variable length
- For most programs, the number of possible symbolic expressions is excessively large
- Unlikely to detect missing paths
- Studies have shown that in general, it is not more effective than the combined use of other methods such as static and dynamic analyses

Types of Errors Detected:
- None, but is an aid for detecting errors. A person must verify the correctness of the output generated by symbolic execution in the same way that output is verified when generated by executing a program over actual values.

References: [ANS104], [EWICS3], [NBS93]

A.30. Test Certification

Description:
The purpose is to ensure that reported test results are the actual finding of the tests. Test related tools, media, and documentation shall be certified to ensure maintainability and repeatability of tests. This technique is also used to show that the delivered software product is identical to the software product that was subjected to V&V.

Advantages:
- Assurance that the test rcsults are accurately presented
- Assurance that the corrected version of product is in compliance with test findings

Disadvantages:

- Often mistaken as a certification of system quality

Type of Errors Detected:
- Incorrect test results reported
- Tests reported that never occurred
- Incorrect version of product shipped

References: [IEEE1012]

A.31. Traceability Analysis (Tracing)

Description:
There are several types of traceability analysis, including requirements trace, design trace, code trace, and test trace. Traceability analysis is the process of verifying that each specified requirement has been implemented in the design / code, that all aspects of the design / code have their basis in the specified requirements, and that testing produces results compatible with the specified requirements.

Advantages:
- Highly effective for detecting errors during design and implementation phases
- Valuable aid in verifying completeness, consistency, and testability of software
- Aids in retesting software when a system requirement has been changed

Disadvantages:
- No significant disadvantages

Types of Errors Detected:

REQUIREMENTS:
- Omitted functions
- Higher-order requirement improperly translated
- Software specification incompatible with other system specifications

DESIGN:
- Omission or misinterpretation of specified requirements
- Detailed design does not conform to top-level design
- Failure to conform to standards

CODE:
- Omission or misinterpretation of specified requirements
- Code does not conform to detailed design
- Failure to conform to standards

TEST:

- Software does not perform functions and produce outputs in conformance with the requirements specification

References: [DUNN], [ANS104], [NBS93]

A.32. Walkthroughs

Description:
A walkthrough is an evaluation technique in which a designer or programmer leads one or more other members of the development team through a segment of design or code, while the other members ask questions and make comments about technique, style, and identify possible errors, violations of development standards, and other problems. Walkthroughs are similar to reviews, but are less formal. Other essential differences include the following:

- Participants are fellow programmers rather than representatives of other functions
- Frequently no preparation
- Scope - standards usually ignored. Successful static analysis results generally not confirmed
- Checklists are rarely used
- Follow-up is often ignored

Advantages:
- Less intimidating than formal reviews
- Identifies the most error-prone sections of the program, so more attention can be paid to these sections during testing
- Very effective in finding logic design and coding errors

Disadvantages:
- Designers or programmers may feel walkthrough is an attack on their character or work

Types of Errors Detected:

- Interface
- Logic
- Data
- Syntax

References: [ANS104], [DUNN], [IEC65A94]

APPENDIX B. ERROR ANALYSIS TECHNIQUES CITED IN SOFTWARE STANDARDS

For [NUREG, NIST204], NIST conducted a survey of software engineering standards and guidelines used in the assurance of the quality of high integrity systems. Many of those documents require or recommend use of software quality assurance and software verification and validation techniques to locate errors in all phases of the software lifecycle. However, these techniques may not be sufficient to detect all errors. NIST extended that study to include other standards and guidelines in order to determine the extent of recommendations, requirements, and guidance for error analysis techniques. The results of this study are presented here.

This study included the examination of approximately 50 documents, which include software standards, draft standards, and guidelines, all selected from a bibliographic search. These documents pertain to high integrity systems, such as safety systems for nuclear power plants, message transmitting devices, medical devices, and other safety-critical software. The list of documents that were reviewed, with the full document titles and corresponding identifiers, are shown in Table B-1.

One objective of this study of standards was to determine whether there is readily available supporting literature on techniques that are required by the standards. The study showed that in most cases, there is not adequate information available in the standards on how to use the techniques, and that a developer would have to rely on library searches (e.g., books, journal articles) or information collected from previous projects. Another objective of this study was to look for consensus in the standards to determine which techniques are generally agreed upon by experts for use.

Results showed that the standards varied in coverage of techniques. Some address techniques in detail, while others only mention them. The study also found that some commonly used techniques like complexity analysis were cited infrequently. Yet, technical literature and availability of complexity metrics tools indicate that complexity analysis is often used to support error detection.

The specific findings are presented in Table B-2 below. For each technique, the table specifies the type, the standards which cite the technique, and the depth of discussion, e.g., whether the technique is mentioned, recommended, defined, or required. Although all of the selected documents were reviewed, information on them does not appear in the table unless they address error analysis in some way.

Only a few techniques are explicitly required. These techniques may be required only under certain circumstances (e.g., reviews are required only during the design phase). Many standards only mention techniques, and do not define, recommend, or require their use. Techniques that are "mentioned" may be examples of appropriate techniques that satisfy certain criteria. A few standards provide extensive information on several techniques, such as IEC65A(Secretariat)122 and AFISC SSH 1-1.

Several documents have appendices which are not part of the standard, but are included with the standard for information purposes. These include ANSI/IEEE-ANS-7-4.3.2-19XX Draft 2, ANSI/ANS-10.4-1987, and ANSI/IEEE Std 1012-1986. The appendices provide short definitions of the selected techniques.

FIPS 101 recommends techniques that can be used in each lifecycle phase. ANSI/IEEE Std 1012-1986 provides similar information in a chart containing optional and required techniques and specifies the lifecycle phases in which they can be used.

Table B-1. List of Reviewed Documents

IDENTIFIER	NUMBER AND TITLE
AFISC SSH 1-1	AFISC SSH 1-1, "Software System Safety," Headquarters Air Force Inspection and Safety Center, 5 September 1985.
ANSI X9.9-1986	ANSI X9.9-1986, "Financial Institution Message Authentication (Wholesale)," X9 Secretariat, American Bankers Association, August 15, 1986.
ANSI X9.17-1985	ANSI X9.17-1985, "Financial Institution Key Management (Wholesale)," X9 Secretariat, American Bankers Association, April 4, 1985.
ANSI/ANS-10.4-1987	ANSI/ANS-10.4-1987, "Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry," American Nuclear Society, May 13, 1987.
ANSI/ASQC A3-1987	ANSI/ASQC A3-1987, "Quality Systems Terminology," American Society or Quality Control, 1987.
ANSI/IEEE Std 730.1-1989	ANSI/IEEE Std 730.1-1989, "IEEE Standard for Software Quality Assurance Plans," Institute of Electrical and Electronics Engineers, Inc., October 10, 1989.
ANSI/IEEE Std 1012-1986	ANSI/IEEE Std 1012-1986, "IEEE Standard for Software Verification and Validation Plans," The Institute of Electrical and Electronics Engineers, Inc., November 14, 1986.
ANSI/IEEE-ANS-7-4.3.2-19XX	ANSI/IEEE-ANS-7-4.3.2-1982, "Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations," American Nuclear Society, 1982. <u>AND</u> ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 2, as of November, 1991.
AQAP-13	AQAP-13, "NATO Software Quality Control System Requirements," NATO, August 1991.
ASME NQA-1-1989 Supplement 17S-1	Supplement 17S-1, ASME NQA-1-1989, "Supplementary Requirements for Quality Assurance Records," The American Society of Mechanical Engineers.
ASME NQA-2a-1990	ASME NQA-2a-1990, "Quality Assurance Requirements for Nuclear Facility Applications," The American Society of Mechanical Engineers, November 1990.
ASME NQA-3-1989	ASME NQA-3-1989, "Quality Assurance Program Requirements for the Collection of Scientific and Technical Information for Site Characterization of High-Level Nuclear Waste Repositories," The American Society of Mechanical Engineers, March 23, 1990.
CAN/CSA-Q396.1.2-89	CAN/CSA-Q396.1.2-89, "Quality Assurance Program for Previous Developed Software Used in Critical Applications," Canadian Standards Association, January 1989.

IDENTIFIER	NUMBER AND TITLE
CSC-STD-003-85	CSC-STD-003-85, "Computer Security Requirements--Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments," Department of Defense, 25 June 1985.
DLP880	DLP880, "(DRAFT) Proposed Standard for Software for Computers in the Safety Systems of Nuclear Power Stations (based on IEC Standard 880)," David L. Parnas, Queen's University, Kingston, Ontario, March, 1991.
DOD 5200.28-STD	DOD 5200.28-STD, "Department of Defense Trusted Computer System Evaluation Criteria," Department of Defense, December 1985.
E.F.T. Message Authentication Guidelines	Criteria and Procedures for Testing, Evaluating, and Certifying Message Authentication Devices for Federal E.F.T. Use," United States Department of the Treasury, September 1, 1986.
FDA/HIMA (DRAFT)	FDA/HIMA, "(DRAFT) Reviewer Guidance for Computer-Controlled Devices," Medical Device Industry Computer Software Committee, January 1989.
FIPS 74	FIPS 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard," U.S. Department of Commerce/National Bureau of Standards, 1981 April 1.
FIPS 81	FIPS 81, "DES Modes of Operation," U.S. Department of Commerce/ National Bureau of Standards, 1980 December 2.
FIPS 46-1	FIPS 46-1, "Data Encryption Standard," U.S. Department of Commerce/National Bureau of Standards, 1988 January 22.
FIPS 101	FIPS 101, "Guideline for Lifecycle Validation, Verification, and Testing of Computer Software," U.S. Department of Commerce/National Bureau of Standards, 1983 June 6.
FIPS 132	FIPS 132, "Guideline for Software Verification and Validation Plans," U.S. Department of Commerce/National Bureau of Standards, 1987 November 19.
FIPS 140 FS 1027	FIPS 140 FS 1027, "General Security Requirements for Equipment Using the Data Encryption Standard," General Services Administration, April 14, 1982.
FIPS 140-1	FIPS 140-1, "Security Requirements for Cryptographic Modules," U.S. Department of Commerce/National Institute of Standards and Technology, 1990 May 2.
Guide to the Assessment of Reliability	"89/97714-Guide to the Assessment of Reliability of Systems Containing Software," British Standards Institution, 12 September 1989.
Guideline for the Categorization of Software	"Guideline for the Categorization of Software in Ontario Hydro's Nuclear Facilities with Respect to Nuclear Safety," Revision 0, Nuclear Safety Department, June 1991.

IDENTIFIER	NUMBER AND TITLE
Guidelines for Assuring Testability (DRAFT)	"(DRAFT) Guidelines for Assuring Testability," The Institution of Electrical Engineers, May 1987.
IEC 880	IEC 880, "Software for Computers in the Safety Systems of Nuclear Power Stations," International Electrotechnical Commission, 1986.
IEC 880 Supplement	45A/WG-A3(Secretary)42, "(DRAFT) Software for Computers Important to Safety for Nuclear Power Plants as a Supplement to IEC Publication 880," International Electrotechnical Commission Technical Committee: Nuclear Instrumentation, Sub-Committee 45A: Reactor Instrumentation, Working Group A3: Data Transmission and Processing Systems, May 1991.
IEC65A(Secretariat)122	IEC/TC65A WG9, "89/33006 DC - (DRAFT) Software for Computers in the Application of Industrial Safety-Related Systems," British Standards Institution, 6 December 1989.
IEC65A(Secretariat)123	IEC/TC65A WG10, "89/33005 DC - (DRAFT) Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems: Generic Aspects, Part 1: General Requirements," British Standards Institution, January 1992.
IFIP WG 10.4	IFIP WG 10.4, "Dependability: Basic Concepts and Terminology," IFIP Working Group on Dependable Computing and Fault Tolerance, October 1990.
Interim Defence Std 00-55	Interim Defence Standard 00-55, "The Procurement of Safety Critical Software in Defence Equipment," Parts 1 and 2, Ministry of Defence, 5 April 1991.
Interim Defence Std 00-56	Interim Defence Standard 00-56, "Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment," Ministry of Defence, 5 April 1991.
ISO 9000	ISO 9000, "International Standards for Quality Management," May 1990.
ITSEC 1.1989	ITSEC 1.1989, "Criteria for the Evaluation of Trustworthiness of Information Technology (IT) Systems," GISA - German Information Security Agency, 1989.
ITSEC 1.2 1991	ITSEC 1.2 1990, "Information Technology Security Evaluation Criteria (ITSEC)," Provisional Harmonised Criteria, June 28, 1990.
Management Plan Documentation Standard	"Management Plan Documentation Standard and Data Item Descriptions (DID)," NASA, 2/28/89.
MIL-HDBK-347	MIL-HDBK-347, "Mission-Critical Computer Resources Software Support," Department of Defense, 22 May 90.
MIL-STD-882B	MIL-STD-882B, "System Safety Program Requirements," Department of Defense, 30 March 1984.

IDENTIFIER	NUMBER AND TITLE
NCSC-TG-005	NCSC-TG-005, "Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria," National Computer Security Center, 31 July 1987.
NCSC-TG-021	NCSC-TG-021, "Trusted Database Management System Interpretation of the Trusted Computer System Evaluation Criteria," National Computer Security Center, April 1991.
NPR-STD-6300	NPR-STD-6300, "Management of Scientific, Engineering and Plant Software," Office of New Production Reactors, U.S. Department of Energy, March 1991.
NSA Spec. 86-16	NSA Spec. 86-16, "Security Guidelines for COMSEC Software Development," National Security Agency, 10 July 1986.
NSWC TR 89-33	NSWC TR 89-33, "Software Systems Safety Design Guidelines and Recommendations," Naval Surface Warfare Center, March 1989.
Ontario Hydro Standard	"Standard for Software Engineering of Safety Critical Software," Rev. 0, Ontario Hydro, December 1990.
P1228 (DRAFT)	P1228, "(DRAFT) Standard for Software Safety Plans," The Institute of Electrical and Electronics Engineers, Inc, July 19, 1991.
Product Specification Documentation	"Product Specification Documentation Standard and Data Item Descriptions (DID)," NASA, 2/28/89.
Programmable Electronic Systems	"Programmable Electronic Systems in Safety Related Applications," Parts 1 and 2, Health and Safety Executive, 1987.
RTCA/DO-178A	RTCA/DO-178A, "Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, March, 1985.
SafeIT	"SafeIT," Volumes 1 and 2, Interdepartmental Committee on Software Engineering, June 1990.
UL 1998	UL 1998, "The Proposed First Edition of the Standard for Safety-Related Software," Underwater Laboratories, August 17, 1990.

Table B-2. Error Analysis Techniques Cited in Reviewed Documents

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
acceptance sampling	statistical technique for quality control	ANSI/ASQC A3-1987	Recommended during installation, defined in glossary

acceptance testing	formal testing to determine if system satisfies acceptance criteria and to enable customer to decide whether or not to accept the product	FIPS 101;
		
		Guidelines for Assuring Testability (DRAFT);
		ISO 9000 (9004);
		FDA/HIMA (DRAFT);
		ANSI/IEEE Std 1012-1986
			Recommended, defined in glossary
			Defined briefly
			Mentioned
			Mentioned
			Defined briefly

algorithm analysis	check that algorithms are correct, appropriate, stable, and meet all accuracy, timing, and sizing requirements	ANSI/ANS-10.4-1987; ANSI/IEEE Std 1012-1986	Defined in appendix only Optional, in appendix
analysis of variance	statistical technique for comparing the means of two or more groups to see if the means differ significantly as a result of some factor/treatment	ISO 9000 (9004)	Mentioned

boundary value analysis (boundary checking)	input test data is selected to create on or around range limits of input or output variable ranges in order to remove errors occurring at parameter limits or boundaries	Guide to the Assessment of Reliability;
		FDA/HIMA (DRAFT);
		NSWC TR 89-33;
		IEC65A(Secretariat)122;
		SafeIT
			Defined briefly
			Recommended (p. 17)
			Required
			Recommended and defined
			Recommended

cause-effect graphing	method of designing test cases by translating a requirements specification into a Boolean graph that links input conditions (causes) with output conditions (effect), analyzing graph to find all combinations of causes that will result in each effect, and identifying test cases that exercise each cause-effect relationship	ANSI/ANS-10.4-1987	Defined in appendix only

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

code inspection	an inspection of the code
	See *inspection*
		E.F.T. Message Authentication Guidelines;
		
		IFIP WG 10.4;
		FDA/HIMA (DRAFT)
			Gives checklist to be used during code inspection
			Mentioned
			Mentioned

compiler checks	static analysis	IFIP WG 10.4	Mentioned
complexity analysis software complexity metrics	static analysis of the complexity of the code using code metrics (e.g. McCabe's, cyclomatic complexity, number of entries/exits per module)	IFIP WG 10.4	Mentioned
constraint analysis	used to ensure that the program operates within the constraints imposed by the requirements, design, and target computer	P1228 (DRAFT)	Defined briefly
control charts	statistical technique used to assess, monitor, and maintain the stability of a process by plotting characteristics of the process on a chart over time and determining whether the variance is within statistical bounds	ANSI/ASQC A3-1987	Mentioned

control flow analysis	check that the proposed control flow is free of problems, e.g., unreachable or incorrect design or code elements	Guidelines for Assuring Testability (DRAFT);
		ANSI/IEEE Std 1012-1986;
		Interim Defence Std 00-55 (Part 2);
		FDA/HIMA (DRAFT)
			Described briefly
			Optional, in appendix
			Defined briefly
			Mentioned

criticality analysis	identifies all software requirements that have safety implications; a criticality level is assigned to each safety-critical requirement	P1228 (DRAFT)	Defined briefly
cusum techniques/quality control charts	statistical technique involving	ISO 9000 (9004)	Mentioned

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

data (use) analysis	used to evaluate the data structure and usage in the code; to ensure that data items are defined and used properly; usually performed in conjunction with logic analysis	Guidelines for Assuring Testability;
		P1228 (DRAFT);
		Interim Defence Std 00-55 (part 2);
		UL 1998
			Described briefly
			Defined briefly
			Defined briefly
			Required

database analysis	check that the database structure and access methods are compatible with the logical design	ANSI/IEEE Std 1012-1986	Optional, in appendix

data flow / corruption analysis	graphical analysis technique to trace behavior of program variables as they are initialized, modified, or referenced when the program executes	IFIP WG 10.4;
		FIPS 101;
		
		FDA/HIMA (DRAFT);
		RTCA/DO-178A;
		ANSI/IEEE Std 1012-1986
			Mentioned
			Recommended during design phase
			Mentioned
			Recommended
			Optional, in appendix

data flow diagrams	a graphical technique that shows how data inputs are transformed to outputs, without detailing the implementation of the transformations	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended

desk checking (code reading)	the reviewing of code by an individual to check for defects and to verify the correctness of the program logic flow, data flow, and output	ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 1;
		Guidelines for Assuring Testability;
		AFISC SSH 1-1
			Defn. briefly in appendix only
			Mentioned
			Defined

error guessing	use of experience and intuition combined with knowledge and curiosity to add uncategorized test cases to the designed test case set in order to detect and remove common programming errors	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended
error seeding	used to ascertain whether a test case set is adequate by inserting known error types into the program, and examining the ability of the program to detect the seeded errors	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
execution flow and timing analysis	See data flow analysis and sizing and timing analysis	RTCA/DO-178A	Recommended
factorial analysis	statistical technique	ISO 9000 (9004)	Mentioned
Fagan inspections	a formal audit on quality assurance documents aimed at finding errors and omissions	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended
failure data models	statistical / probabilistic technique for assessing software reliability	Guide to the Assessment of Reliability	Defined

finite state automata/ finite state machines/ state transition diagram	modelling technique that defines the control structure of a system and is used to perform static analysis of system behavior	IFIP WG 10.4;
		FIPS 140-1;
		IEC65A(Secretariat)122;
		SafeIT
			Mentioned
			Required
			Recommended and defined
			Recommended

(formal) mathematical modelling	modelling technique, static analysis	ANSI/ASQC A3-1987;
		IEC65A(Secretariat)122;
		SafeIT;
		Management Plan Documentation Standard
			Mentioned
			Recommended and defined
			Recommended
			Mentioned

formal analysis (formal proofs, formal verification, formal testing)	use of rigorous mathematical techniques to analyze the algorithms of a solution for numerical properties, efficiency, and/or correctness
	
	formal verification - proving mathematically that a program satisfies its specifications
		ITSEC 1991
		ANSI/ANS-10.4-1987;
		ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 2;
		Guide to the Assessment of Reliability;
		Guidelines for Assuring Testability (DRAFT);
		SafeIT;
		P1228 (DRAFT);
		IEC65A(Secretariat)122;
		NCSC-TG-005
			Defined, required p. 98
			Defined in appendix only
			Defn. briefly in appendix only
			Defined briefly
			Defined
			Recommended
			Mentioned
			Recommended and defined
			Defined

frequency distribution	a graph that displays the frequency of a variable as a bar, whose area represents the frequency	ANSI/ASQC A3-1987	Mentioned

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

functional testing	application of test data derived from the specified functional requirements without regard to the final program structure	ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 2;
		IFIP WG 10.4;
		Guide to the Assessment of Reliability;
		FIPS 101;
		ANSI/ANS-10.4-1987;
		IEC65A(Secretariat)122;
		SafeIT;
		NCSC-TG-005;
		FDA/HIMA (DRAFT)
			Defined in appendix only
			Mentioned
			Defined briefly
			Defined in glossary
			Defined in appendix only
			Recommended and defined
			Recommended
			Mentioned, defined
			Mentioned

global roundoff analysis of algebraic processes (algorithmic analysis)	determines how rounding errors propagate in a numerical method for permissible sets of input data; one type of algorithm analysis	ANSI/ANS-10.4-1987	Defined, in appendix only
information flow analysis	extension of data flow analysis	Guidelines for Assuring Testability; Interim Defence Std 00-55 (part 2)	Described briefly Defined briefly

inspection	software requirements, design, code, or other products are examined by a person or group other than the author to detect faults, violations of standards, and other problems	IFIP WG 10.4;
		Guide to the Assessment of Reliability;
		FIPS 101;
		
		ANSI/ANS-10.4-1987;
		ANSI/ASQC A3-1987;
		FDA/HIMA (DRAFT)
			Mentioned
			Defined briefly
			Defined in glossary, recommended for design phase
			Defined in appendix only
			Defined briefly
			Defined in glossary

integrated critical path analysis		MIL-STD-882B	Mentioned

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
integration testing	orderly progression of testing in which software elements, hardware elements, or both, are combined and tested until all intermodule communication links have been integrated	FIPS 101; FDA/HIMA (DRAFT)	Defined in glossary Mentioned

interface analysis (testing)	checking that intermodule communication links are performed correctly; ensures compatibility of program modules with each other and with external hardware and software	IEEE Std 1012-1986;
		FIPS 101;
		
		MIL-HDBK-347;
		NSWC TR 89-33;
		P1228 (DRAFT)
			Required
			Recommended during design phase, defined in glossary
			Required
			Required
			Defined briefly

logic analysis	evaluates the sequence of operations represented by the coded program and detects programming errors that might create a hazard	P1228 (DRAFT)	Defined briefly
Markov models	graphical technique used to model a system with regard to its failure states in order to evaluate the reliability, safety or availability of the system	Interim Defence Standard 00-56 . IEC65A(Secretariat)122	Mentioned Recommended and defined
measure of central tendency and dispersion	statistical technique for quality control	ANSI/ASQC A3-1987	Mentioned
module functional analysis	verifies the relevance of software modules to safety-critical requirements, assuring that each requirement is adequately covered and that a criticality level is assigned to each module	P1228 (DRAFT)	Defined briefly
mutation analysis (testing)	process of producing a large set of program versions, each derived from a trivial alteration to the original program, and evaluating the ability of the program test data to detect each alteration	IFIP WG 10.4; ANSI/ANS-10.4-1987	Mentioned In appendix only
NO-GO (path) testing	type of failure testing	UL 1998; NSWC TR 89-33	Required Required

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
peer review	software requirements, design, code, or other products are examined by persons whose rank, responsibility, experience, and skill are comparable to that of the author	ANSI/ANS-10.4-1987	Defined in appendix only
performance modelling	modelling of the system to ensure that it meets throughput and response requirements for specific functions, perhaps combined with constraints on use of total system resources	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended
periodic testing	security testing	NCSC-TG-005	Mentioned, defined
penetration testing	security testing in which the penetrators attempt to circumvent the security features of a system	NCSC-TG-005	Mentioned, defined in glossary

Petri nets	graphical technique used to model relevant aspects of the system behavior and to assess and improve safety and operational requirements through analysis and re-design	IFIP WG 10.4;
		AFISC SSH 1-1;
		MIL-STD-882B;
		P1228 (DRAFT);
		IEC65A(Secretariat)122
			Mentioned
			Defined
			Mentioned
			Mentioned
			Recommended and defined

proof of correctness	use of techniques of mathematical logic to infer that a relation between program variables assumed true at program entry implies that another relation between them holds at program exit	IFIP WG 10.4;
		FIPS 101;
		Guidelines for Assuring Testability;
		IEC65A(Secretariat)122;
		AFISC SSH 1-1
			Mentioned
			Defined in glossary
			Defined briefly
			Recommended
			Defined

protocol testing	security testing	NCSC-TG-005	Mentioned and defined

prototyping/animation	check the feasibility of implementing a system against the given constraints by building and evaluating a prototype with respect to the customer's criteria;
	dynamic test of code through execution
		Interim Defence Std 00-55 (Part 2);
		Guide to the Assessment of Reliability;
		Management Plan Documentation Standard;
		IEC65A(Secretariat)122;
		SafeIT
			Defined
			Defined briefly
			Defined briefly in glossary
			Recommended and defined
			Recommended

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
regression technique	statistical technique for determining from actual data the functional relationship between 2 or more correlated variables	ISO 9000 (9004); ANSI/ASQC A3-1987	Mentioned Recommended

regression analysis / testing	selective retesting of a product to detect faults introduced during modification, to verify that modification had no unintended side effects and that the modified software still meets specified requirements	Guide to the Assessment of Reliability;
		FIPS 101;
		ANSI/ANS-10.4-1987;
		MIL-HDBK-347;
		P1228 (DRAFT);
		ANSI/IEEE Std 1012-1986;
		NSWC TR 89-33
			Defined briefly
			Defined in glossary
			Defined in appendix only
			Required
			Mentioned
			Optional, in appendix
			Required

reliability block diagram	technique for modelling the set of events that must take place and conditions which must be fulfilled for a successful operation of a system or task	Guide to the Assessment of Reliability (part 6 of BS 5760);
		Interim Defence Std 00-56;
		IEC65A(Secretariat)122
			Mentioned
			
			Mentioned
			Recommended and defined

reliability growth models	a statistical model used to predict the current software failure rate and hence the operational reliability	Guide to the Assessment of Reliability	Mentioned

requirements analysis (review)	involves translation of informal prose requirements into formal representation in order to identify aspects of the requirements needing clarification or further definition	ASME NQA-2a-1990;
		FDA/HIMA (DRAFT);
		FIPS 101
			Required
			Mentioned
			Defined in Appendix B

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

reviews and audits
 	
Includes: formal review, software requirements review, (preliminary) design review, critical design review, managerial review, contract review, functional audit, physical audit, in-process audit, SCM plan review, etc.
	meeting at which software requirements, design, code or other product is presented to the user, sponsor, or other interested parties for comment and approval, often as a prerequisite for concluding a given phase of the software lifecycle	ANSI/IEEE Std 1012-1986;
		ISO 9000 (9000-3);
		ISO 9000 (9001);
		E.F.T. Message Authentication Guidelines;
		Guide to the Assessment of Reliability;
		FIPS 101;
		Guidelines for Assuring Testability;
		ANSI/ANS-10.4-1987;
		Management Plan Documentation Standard;
		CAN/CSA-Q396.1.2-89;
		ASME NQA-2a-1990;
		ANSI/ASQC A3-1987;
		FDA/HIMA (DRAFT);
		IEC65A(Secretariat)122;
		SafeIT;
		ANSI/IEEE 730.1-1989
			Mgmt. review defined, req'd
			Recommended for design
			Mgmt., contract review req'd
			Mentioned
			Defined briefly
			Recommended during reqmts.
			Mentioned
			Defined in appendix only
			Defined briefly in glossary
			Defined briefly
			Required
			Defined briefly
			Defined in glossary
			Recommended and defined
			Recommended
			Described briefly, gives documentation requirements

safety evaluation/risk analysis	statistical analysis	ISO 9000 (9004)	Mentioned

simulation (simulation analysis)	simulate critical aspects of the software environment to analyze logical or performance characteristics that would not be practical to analyze manually
	use of an executable model to examine the behavior of the software
		ANSI/IEEE Std 1012-1986;
		FIPS 101;
		Guidelines for Assuring Testability (DRAFT);
		IEC65A(Secretariat)122;
		SafeIT;
		ANSI/IEEE Std 1012-1986
			Optional, in glossary
			Defined in glossary
			Described briefly
			Recommended and defined
			Recommended
			Optional, in appendix

sizing and timing analysis	determine whether the program will satisfy processor size and performance requirements allocated to software	P1228 (DRAFT); ANSI/IEEE Std 1012-1986	Defined Optional, in appendix

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS
specification analysis	evaluates each safety-critical requirement with respect to quality attributes (e.g., testability)	P1228 (DRAFT); FDA/HIMA (DRAFT)	Defined briefly Mentioned

sneak circuit analysis	graphical technique which relies on the recognition of basic topological patterns in the software structure, used to detect an unexpected path or logic flow which, under certain conditions, initiates an undesired function or inhibits a desired function	AFISC SSH 1-1;
		MIL-STD-882B;
		IEC65A(Secretariat)122;
		SafeIT;
		P1228 (DRAFT)
			Defined
			Mentioned
			Recommended and defined
			Recommended
			Mentioned

standards audit	check to ensure that applicable standards are used properly	FIPS 101	Recommended during code phase, defined in glossary
statistical sampling inspection	method for evaluating a product by testing random samples	ISO 9000 (9004)	Mentioned
statistical testing	type of testing where the test patterns are selected according to a defined probability distribution on the input domain	IFIP WG 10.4	Mentioned, defined in glossary

stress testing	testing a product using exceptionally high workload in order to show that it would stand normal workloads easily	NCSC-TG-005;
		IEC65A(Secretariat)122;
		FDA/HIMA (DRAFT)
			Mentioned, defined
			Defined
			Mentioned

structural analysis (testing)	uses automated tool that seeks and records errors in the structural makeup of a program	ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 2
		IFIP WG 10.4;
		Guide to the Assessment of Reliability;
		AFISC SSH 1-1
			Defined in appendix only
			Mentioned, defined in glossary
			Defined briefly
			Defined

structure diagrams	a notation complementing data flow diagrams, shows hierarchy of system as a tree; shows relationships between program units without including information about the order of execution of these units	IEC65A(Secretariat)122; SafeIT	Recommended and defined Recommended

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

symbolic evaluation (execution)	program execution is simulated using symbols rather than actual numerical values for input data, and output is expressed as logical or mathematical expressions involving these symbols	IFIP WG 10.4;
		Guide to the Assessment of Reliability;
		FIPS 101;
		
		ANSI/ANS-10.4-1987
			Mentioned, defined in glossary
			Defined briefly
			Recommended for code, defined in glossary
			Defined in appendix only

system test	testing an integrated hardware and software system to verify that the system meets it specified requirements	FIPS 101; FDA/HIMA (DRAFT)	Defined in glossary Mentioned
test certification	check that reported test results are the actual findings of the tests. Test-related tools, media, and documentation should be certified to ensure maintainability and repeatability of tests	ANSI/IEEE Std 1012-1986	Optional, in appendix
tests of significance	statistical technique	ISO 9000 (9004); ANSI/ASQC A3-1987	Mentioned Mentioned
tracing (traceability analysis)	cross-checking between software products (e.g. design and code) of different lifecycle phases for consistency, correctness, and completeness	ANSI/ANS-10.4-1987; ANSI/IEEE Std 1012-1986	Defined in appendix only Required
unit testing (module testing)	testing of a module for typographic, syntactic, and logical errors, for correct implementation of its design, and satisfaction of its requirements	FIPS 101 FDA/HIMA (DRAFT)	Defined in glossary, in appd. Mentioned

TECHNIQUE	DESCRIPTION	STANDARDS	STATUS

walkthroughs	evaluation technique in which a designer or programmer leads one or more other members of the development team through a segment of design or code, while the other members ask questions and comment about technique, style, and identify possible errors, violations of standards, and other problems
	
	evaluation technique in which development personnel lead others through a structured examination of a product
		ANSI/IEEE-ANS-7-4.3.2-19XX, Draft 2
		IFIP WG 10.4;
		Guide to the Assessment of Reliability;
		FIPS 101;
		ANSI/ANS-10.4-1987;
		FDA/HIMA (DRAFT);
		RTCA/DO-178A;
		MIL-STD-882B;
		IEC65A(Secretariat)122;
		SafeIT;
		AFISC SSH 1-1;
		ANSI/IEEE Std 1012-1986
			Defn. briefly in appendix only
			Mentioned
			Defined briefly
			Defined in glossary
			Defined in appendix only
			Mentioned
			Recommended
			Mentioned
			Recommended and defined
			Recommended
			Defined
			Optional, in appendix

☆ U.S. GOVERNMENT PRINTING OFFICE: 1993—3 4 1 -8 3 2 / 8 3 0 3 8