IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

Circuits and Devices

Communications Technology

Computer

Sponsored by the
Software Engineering Technical Subcommittee of the
Technical Committee on Software Engineering of the
IEEE Computer Society

Electromagnetics and Radiation

Energy and Power

Industrial Applications

Signals and Applications

Standards Coordinating Committees

IEEE Std 982.2-1988

Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

June 12, 1989

SH12559

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

Sponsor

Software Engineering Technical Subcommittee
of the
Technical Committee on Software Engineering
of the
IEEE Computer Society

Approved September 27, 1988

IEEE Standards Board

Corrected Edition
June 12, 1989

NOTE: A black bar has been placed in the margin next to each correction to aid in identifying the changes made to this printing.

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

Foreword

(This Foreword is not a part of IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.)

This guide provides the underlying concepts and motivation for establishing a measurement process for reliable software, utilizing IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software. This guide contains information necessary for application of measures to a project. It includes guidance for the following:

(1) Applying product and process measures throughout the software life cycle, providing the means for continual self-assessment and reliability improvement;
(2) Optimizing the development of reliable software, beginning at the early development stages, with respect to constraints such as cost and schedule;
(3) Maximizing the reliability of software in its actual use environment during the operation and maintenance phases;
(4) Developing the means to manage reliability in the same manner that cost and schedule are managed.

The guide is intended for design, development, evaluation (eg, auditing or procuring agency) and maintenance personnel; software quality and software reliability personnel; and operations and acquisition support managers. It is organized to provide input to the planning process for reliability management.

This document contains six sections and one appendix.

Section 1, "Scope and References," establishes the goals and boundaries of the guide.

Section 2, "Definitions," serves as a central location for key terms used throughout the body of the document.

Section 3, "Measures to Produce Reliable Software," contains guidance for measure selection and application.

Section 4, "Measure Organization and Classification," provides both functional and software life cycle taxonomies of the measures.

Section 5, "Framework for Measures," refines the measurement process to nine stages for effective implementation.

Section 6, "Errors, Faults, and Failures Analysis for Reliability Improvement," provides further definition of the interrelationship of these concepts.

Appendix A, "Measures for Reliable Software," contains the standard set of measures.

The goal of IEEE Std 982.2-1988, IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software is to generate a dialogue on the use and effectiveness of selected measures. This dialogue should result in the evolution of existing measures and the development of new measures that evaluate and promote software reliability. In this way, the body of knowledge of and experience with software reliability can be improved and expanded.

This is one of a series of integrated IEEE Software Engineering Standards. This guide is consistent with ANSI/IEEE Std 729-1983, Glossary of Software Engineering Terminology; ANSI/IEEE Std 730-1984, IEEE Standard for Software Quality Assurance Plans; ANSI/IEEE Std 828-1983, IEEE Standard for Software Configuration Management Plans; and ANSI/IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans. This guide may be applied either independently or in conjunction with those standards.

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

F. Ackerman
W. Boll
J. Bowen
P. Bright
F. Buckley
H. Carney
J. Chung
F. Coallier
P. Denny
J. Dobbins
D. Doty
R. Dwyer
W. Ellis
W. Farr
D. Favor
J. Fendrich
G. Fields
J. Forster
D. Gelperin
S. Glickman
S. Gloss-Sloer
L. Good
D. Gustavson
V. Haas
C. Hay
H. Hecht
L. Kaleda
T. Kurihara
G. Larsen
F. C. Lim
B. Lindberg
M. Lipow
B. Livson
P. Marriott
R. Martin
J. McCall
J. Mersky
W. Murch
J. Navlakha
D. Nickle
D. Peercy
W. Perry
P. Petersen
D. Pfeiffer
R. Poston
P. Powell
N. Schneidewind
W. Schnoege
R. Schueppert
R. Shillato
D. Siefert
D. Simkins
R. Singh
V. Srivastava
A. Sukert
K. Tai
R. Thibodeau
P. Thompson
D. Town
H. Trochesset
R. van Tilburg
P. Wolfgang
T. Workman
N. Yopconka

The Software Reliability Measurement Working Group had the following membership:

B. Andrews
S. Beason
R. Berlack
M. Berry
N. Beser
J. Bieman
J. Blackman
J. Bowen
M. Bruyere
F. Buckley
N. Chu
G. Chisholm
G. Clow
J. Corrinne
W. Covington
W. Daney
B. Daniels
F. DeKalb
I. Doshay
E. Dunaye
F. Fisk
A. Florence
D. Frevert
D. Gacke
W. Goss
S. Gray
F. Gray
V. Haas
C. Harrison
S. Hartman
L. Hearn
H. Hecht
R. Hood
J. Horch
D. Hurst
N. Johnson, Jr
R. Kenett
R. Kessler
R. Kumar
M. Landes
R. Lane
J. Latimer
L. Lebowitz
M. Lipow
C. McPherson
A. Moseley
S. Nemecek
K. Nidiffer
K. Oar
R. Panchal
R. Pikul
E. Presson
R. Prudhomme
H. Richter
M. Richter
L. Sanders
R. Sandborgh
H. Schock
D. Simkins
O. Smith
R. Spear
E. Soistman
L. Sprague
A. Stone
D. Sudbeck
R. Sulgrove
D. Town
R. Wamser
J. Wang
D. Winterkorn
J. Wujek
B. Zamastil

The Software Reliability Measurement Working Group Steering Committee had the following members:

James Dobbins, Chairman

Ray Leber, *Cochairman*

Ted Workman, *Cochairman*

P. Bright
A. Cicu
C. Cucciati
W. Ellis
W. Farr
S. Glickman
L. Good
N. Hall
P. Marriott
R. Martin
W. Murch
P. Powell
F. Salvia
D. Siefert
R. Singh
V. Srivastava
H. Trochesset
W. Wilson

When the IEEE Standards Board approved this standard on September 27, 1988, it had the following membership:

Donald C. Fleckenstein, *Chairman* **Marco Migliaro**, *Vice Chairman*

Andrew G. Salem, *Secretary*

Arthur A. Blaisdell	John W. Horch	L. Bruce McClung
Fletcher J. Buckley	Jack M. Kinn	Donald T. Michael*
James M. Daly	Frank D. Kirschner	Richard E. Mosher
Stephen R. Dillon	Frank C. Kitzantides	L. John Rankine
Eugene P. Fogarty	Joseph L. Koepfinger*	Gary S. Robinson
Jay Forster*	Irving Kolodny	Frank L. Rose
Thomas L. Hannan	Edward Lohse	Helen M. Wood
Kenneth D. Hendrix	John E. May, Jr.	Karl H. Zaininger
Theodore W. Hissey, Jr.	Lawrence V. McCall	Donald W. Zipse

*Member Emeritus

Comments on this guide are welcome and should be addressed to:

James Dobbins
Director, Software Quality Engineering
AMS, Inc.
1525 Wilson Boulevard, 3rd Floor
Arlington, VA 22209

Contents

Figures

Tables

Appendix A
Measures for Reliable Software

APPENDIX FIGURES

APPENDIX TABLES PAGE

List of Symbols

Symbol	Meaning	Measures Used	Primitive/ Metric
A	(1) Number of arrivals during time period T	37	P
	(2) Reliability growth factor	26	M
$A_{existing}$	Number of ambiguities in a program remaining to be eliminated	6	P
$A(t)$	System availability	39	M
A_{tot}	Total number of ambiguities identified	6	P
α	Test accuracy	36	M
AQ	Average queue length	37	M
AS	Average service time	37	P
B	Software science measure (number of errors)	14	M
B_1	Number of functions not satisfactorily defined	35	P
B_2	Number of functions	35	P
B_3	Number of data references not having an origin	35	P
B_4	Number of data references	35	P
B_5	Number of defined functions not used	35	P
B_6	Number of defined functions	35	P
B_7	Number of referenced functions not defined	35	P
B_8	Number of referenced functions	35	P
B_9	Number of decision points not using all conditions, options	35	P
B_{10}	Number of decision points	35	P
B_{11}	Number of condition options without processing	35	P
B_{12}	Number of condition options	35	P
B_{13}	Number of calling routines with parameters not agreeing with defined parameters	35	P
B_{14}	Number of calling routines	35	P
B_{15}	Number of condition options not set	35	P
B_{16}	Number of set condition options having no processing	35	P
B_{17}	Number of set condition options	35	P
B_{18}	Number of data references having no destination	35	P
β	Observed hardware failure rate	39	P
C	Complexity metric (static, dynamic, cyclomatic)	15, 16	M
C_i	Complexity for program invocation and return along each edge e_i as determined by the user	15	P
CE	Cause and effect measure	6	M
CM	Completeness measure	35	M
D	Software science measure (program difficulty)	14	M
d_k	Complexity for allocation of resource k as determined by the user	15	P

Symbol	Meaning	Measures Used	Primitive/ Metric
D_i	Total number of unique defects detected during the ith design or code inspection process or the ith life cycle phase	2, 8	P
DD	Defect density	2	M
*DE*s	Decomposition elements	23	P
DI	Defect index	8	M
DSM	Design structure metric	19	M
datain	Number of data structures from which the procedure retrieves data	25	P
dataout	Number of data structures that the procedure updates	25	P
E	(1) Number of edges (2) Software science measure (effort)	15, 16, 17 14	P M
e_i	Number of entry points for the ith module	13	P
F or F_i	Total number of unique faults found in a given time interval resulting in failures of a specified severity level	1	P
f or f_i	Total number of failures of a given severity level in a given time interval	3, 20, 21, 27, 31	P
F_d	Fault density	1	M
f_{ij}	Frequency execution of intermodule transfer from module i to j	38	P
F_a	Number of software functions (modules) in the current delivery that are additions in the current delivery	10	P
F_c	Number of software functions (modules) in the current delivery that include internal changes from a previous delivery	10	P
FD	Fault days metric	4	M
FD_i	Fault days for the ith fault	4	P
F_{del}	Number of software functions (modules) in the previous delivery that are deleted in the current delivery	10	P
FE	Number of the software functional (modular) requirements for which all test cases have been satisfactorily completed	5	P
F_{it}	Total number of faults detected to date in software integration testing	29	P
F_{pit}	Number of faults detected prior to software integration testing	29	P
FT	Total number of software functional test requirements (modular)	5	P
Γ	Average rate of jobs	37	M
Υ	Observed hardware repair rate	39	P
I	Total number of inspections to date	2, 11	P
IFC	Information flow complexity	25	M
J	Total number of jobs completed	37	P
K	Number of resources	15	P

Symbol	Meaning	Measures Used	Primitive/ Metric
k	Number of runs for the specified period of time	18	P
KSLOC	Number of source lines of executable code and nonexecutable data declarations in thousands	1, 2, 8	P
KSLOD	Number of source lines of design statements in thousands that have been inspected to date	2, 8	P
L	Software science measure (observed program length)	14	M
l	Software science measure (program vocabulary)	14	M
Ll	Software science measure (program level)	14	M
$\lambda(t)$	Failure rate function	31	M
λ	Observed software failure rate	39	P
length	Number of source statements in a procedure	25	P
lfi	Local flows into a procedure	25	P
lfo	Local flows from a procedure	25	P
M	Manhours per major defect detected	11	M
M_i	Number of medium defects found	8	P
m_i	Number of entry and exit points for module i	13	M
MD	Number of modules	38	P
$ME(i)$	Measure effectiveness for the ith issue	34	P
M_{it}	Number of modules integrated	29	M
M_T	Number of software functions (modules) in the current delivery	10	M
M_{tot}	Total number of modules in final configuration	29	M
MTTF	Mean-time-to-failure	30	M
μ_i	Observed software fault correction rate with i faults in the software	39	P
N	Number of nodes	15, 16, 17	P
nc	Number of correct runs in a given test sample	18, 26	P
NF	Total number of faults within a program	29, 39	M
NF_{rem}	Total number of faults remaining within a program	22, 29	M
NF_{rem} (%)	Percentage of faults remaining within a program	36	M
n_F	Number of faults found that were not intentionally seeded	22	P
NR	Number of runs (or test cases) in a given test sample	18, 26	P
NS	Total number of stages	26	P
N_s	Number of seeded faults	22, 36	P
n_s	Number of seeded faults found	22	P
$N1$	Total number of occurrences of the operators in a program	14	P
$n1$	Number of distinct operators in a program	14	P

Symbol	Meaning	Measures Used	Primitive/ Metric
$N2$	Total number of occurrences of the operands in a program	14	P
$n2$	Number of distinct operands in a program	14	P
N_1	Number of errors detected using SVDs due to inconsistencies	23	P
N_2	Number of errors detected using SVDs due to incompleteness	23	P
N_3	Number of errors detected using SVDs due to misinterpretation	23	P
P	Probability measure over the sample space	18	P
p_n	Probability of repairing a hardware failure correctly	39	P
P_i	(1) Probability that the ith run is selected from the sample space	18, 38	P
	(2) Probability that the ith process that can be generated in a user environment is selected	39	
P_s	Probability of correcting a software fault when detected	39	P
$P_{N,n}(t)$	System upstate probabilities	39	P
PI_i	Phase index metric for the ith life cycle phase	8	M
PL	Purity level	21	M
PS	Product size	8	P
$P1$	Total number of modules in the program	19	P
$P2$	Number of modules dependent on the input or output	19	P
$P3$	Number of modules dependent on prior processing	19	P
$P4$	Number of database elements	19	P
$P5$	Number of nonunique database elements	19	P
$P6$	Number of database segments (partition of the state)	19	P
$P7$	Number of modules not single entrance/single exit	19	P
Q	Queue length distribution	37	P
q_i	Probability that P_i will be generated in a user environment	38	P
R	(1) Total response time for each functional job	37	M
	(2) Reliability of the system	38	M
R_i	Reliability of the ith module	38	P
R_k	Run reliability at a specified stage	18, 26	M
$R(t)$	Reliability function	31	M
r_i	Flag to denote if the ith process generates the correct software system output	38	P
r_{ki}	Resource status array indicator flags	15	P
RG	Number of regions	16, 17	P
$RK(i)$	Risk factor for the ith measure	34	M
RT	Response time distribution	37	P
$R1$	Number of requirements met by the architecture	7	P

Symbol	Meaning	Measures Used	Primitive/ Metric
$R2$	Number of original requirements	7	P
S	Sample space of possible input patterns and states	18	P
S_i	Point from the sample space of possible inputs	18	P
s_i	Number of serious defects found	8, 11	P
SB	Total amount of time server is busy	37	P
SE	Server's efficiency	37	M
SN	Number of splitting nodes	16, 17	P
SMI	Software maturity index	10	M
ST	Service time distribution	37	P
T	(1) Time period over which measurements are made (2) Time	37 14	P M
T_i	Number of trivial defects found	8	P
t_i	Observed times between failures of a given severity level	20, 21, 27 28, 30, 31	P
TC	Test coverage	24	M
TM	Traceability measure	7	M
T_1	Time expended by the inspection team in preparation for design or code inspection meeting	11	P
T_2	Time expended by the inspection team in conduct of a design or code inspection meeting	11	P
U	Utilization	37	P
V	Software science measure (program volume)	14	M
VR	Number of requests per functional job for each server during time period T	37	P
W_i	Weighting distribution	8, 19, 34	P
WT	Waiting time distribution	37	P
X	Throughput	37	P
x_i	Number of exit points for the ith module	13	P

NOTE: P stands for primitive
M stands for derived metric

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

1. Scope and References

1. Scope. This guide provides the conceptual insights, implementation considerations, and assessment suggestions for the application of IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software [2].[1] The Dictionary, by its very nature, is limited in what it may contain. While this is necessary for standardization, abbreviation of detail might be detrimental to the most effective use of the Dictionary. This guide is written to provide a bridge between IEEE Std 982.1-1988 and the user, so that full benefit may be derived from the application of the standard to a software project.

1.2 References. This guide shall be used in conjunction with the following publications:

[1] ANSI/IEEE 729-1983, IEEE Standard Glossary of Software Engineering Terminology.[2]

[2] IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.[3]

2. Definitions

Most of the definitions listed below can be found in ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology [1]. The definitions of the terms "concept phase," "defect," "measure," "primitive," "software reliability management" are not present in ANSI/IEEE Std 729-1983, and establish meaning in the context of this guide. The acronym ISO (International Organization for Standardization) indicates that the definition is also accepted by ISO.

concept phase. The period of time in the software life cycle during which system concepts and objectives needed by the user are identified and documented. Precedes the requirements phase.

defect. A product anomaly. Examples include such things as (1) omissions and imperfections found during early life cycle phases and (2) faults contained in software sufficiently mature for test or operation. See also "fault."

error. Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification (see ANSI/IEEE Std 729-1983 [1]).

failure. (1) The termination of the ability of a functional unit to perform its required function. (ISO; ANSI/IEEE Std 729-1983 [1]). (2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is encountered [1].

fault. (1) An accidental condition that causes a functional unit to fail to perform its required function. (ISO; ANSI/IEEE Std 729-1983 [1]). (2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonymous with bug [1].

measure. A quantitative assessment of the degree to which a software product or process possesses a given attribute.

primitive. Data relating to the development or use of software that is used in developing measures or quantitative descriptions of software.

---

[1] The numbers in brackets correspond to those of the references in 1.2; numbers in brackets preceded by an "A" correspond to references in the Appendix.

[2] ANSI/IEEE publications are available from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018, or from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08854-1331.

[3] IEEE publications are available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08854-1331.

Primitives are directly measurable or countable, or may be given a constant value or condition for a specific measure. Examples include error, failure, fault, time, time interval, date, number of noncommentary source code statements, edges, nodes.

software reliability. The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [1].

software reliability management. The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources (cost), schedule, and performance.

3. Measures to Produce Reliable Software

This guide addresses two major topics. The first is the rationale for the document IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software. The guide presents the measure selection criteria. It address these questions: "To whom is the standard directed?" "When should the standard be applied?" and "Why was the standard created?"

The second topic is the application of these measures to product, project, and industry. For each measure the guide provides further definition, examples, interpretation, and references. The questions addressed include: "How can the standard be used effectively?" "How should results be interpreted?" and "How should objectives be set?"

The goal of the Dictionary is to support software developers, project managers, and system users in achieving optimum reliability levels in software products. It was designed to address the needs of software developers and customers who are confronted with a plethora of models, techniques, and measures in the literature, but who lack sufficient guidance to utilize them effectively. The standard addresses the need for a uniform interpretation of these and other indicators of reliability.

The Dictionary assumes an intimate relationship between the reliability of a product and the process used to develop that product. The reliable product provides confirmation of a successful process; the unreliable product provides a lesson for process change. It is in this general spirit that the measures selected give visibility to both process and product so that the essential facts necessary for process evaluation and change are available.

The measures were selected to provide information throughout the life cycle of a product. They can be used to make optimum operational reliability levels and to provide a *post factum* measurement of reliability levels achieved. The basic goal is to provide the elements of a measurement program that support a constructive approach for achieving optimum reliability of the end product.

The selected measures are related to various product and process factors that may have an impact on software reliability and that are observable throughout the entire life cycle. The Dictionary provides, even in the early life cycle phases, a means for continual assessment of the evolving product and the process. Through early reliability assessment, the Dictionary supports constructive optimization of reliability within given performance, resource, and schedule constraints.

The Dictionary focuses on measures of the potential causes of failure throughout the software life cycle, rather than just on measures of the effect of poor reliability of nearly completed products. The intent is to produce software that is reliable, rather than just an estimate of the failure-freeness of a nearly completed and possibly unreliable product. Both the traditional approach of measuring reliability and the constructive approach of building in reliability are placed in context in this document.

The primitives to be used and the method of computation are provided for each measure. The standard calibrates the rulers, the measurement tools, through the use of common units and a common language. It promotes the use of a common database in the industry. Through commonality the standard provides the justification for building effective tools for the measurement process itself.

The standard and this guide are intended to serve as the foundation on which researchers and practitioners can build consistent methods. These documents are designed to assist management in directing product development and support toward specific reliability goals. The purpose is to provide a common set of definitions through which a meaningful exchange of data and evaluations can occur. Successful application of the

measures is dependent upon their use as intended in the specified environments.

The future study of extensive sets of basic data collected through the application of this standard will make it feasible to understand causal relationships, if any, between reliability indicators obtained during the process and operational reliability of a completed product. It will also promote the refinement of existing models and the development of new models based on this consistent data.

3.1 Constructive Approach to Reliable Software. An analysis of the relationship between the software development and maintenance processes and software reliability is provided. The purpose is to clarify the criteria used in the selection of measures.

Software reliability measurement is the determination of the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [1].

3.1.1 Constructive Approach. Reliability is an estimation of system failure-freeness. System failures that are caused by software are a valid concern of the software community. A constructive approach to reliable software seeks to remove the root causes of this class of system failures (Fig 3.1.1-1) through software development and support processes that promote fault avoidance, early fault detection, appropriately prompt removal, and system-designed fault tolerance (Fig 3.1.1-2).

The analysis of the errors, faults, and failures from previous development and support processes can lead to improved future processes. While the exact functional relationships are not proven, it is through experience that the majority of failures are related to their origins. Examples include the following:

(1) Incompletely defined user needs and requirements
(2) Omissions in the design and coding process
(3) Improper usage of the system
(4) Excessive change activity

The error, fault, and failure analysis is detailed further in Section 6.

A process strategy to achieve reliable software includes activities, methods, tools, and measurements. Figure 3.1.1-3 shows the process strategy

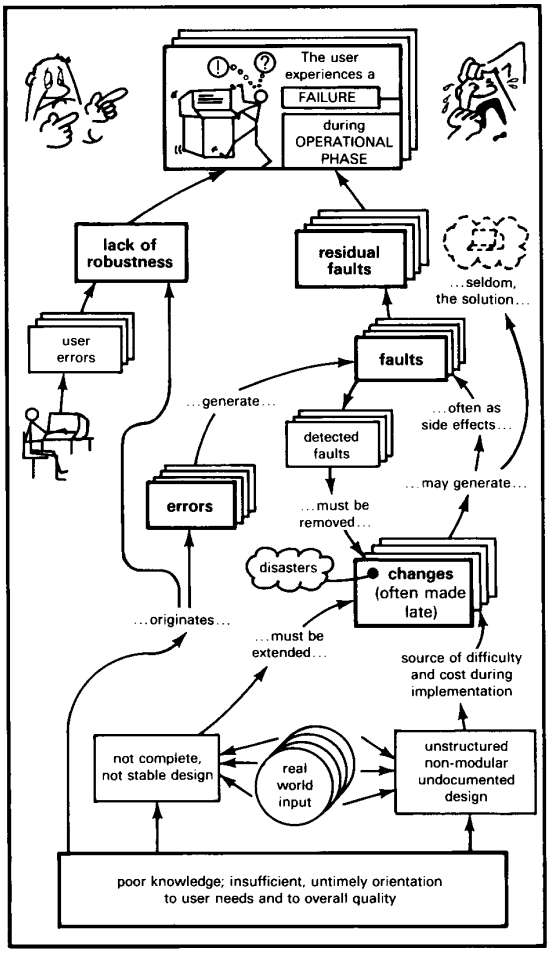


Fig 3.1.1-1
Causes of Failures

and the integrated role of measurements as the control window through which the quality of the product and of the process are monitored.

While Fig 3.1.1-3 is in large part self-explanatory, it is worthwhile to emphasize the major drivers for the process strategy. The strategy does not itself contain new techniques or new methodologies. It is a collection of known concepts, activities, and techniques, that can be used to drive a project toward better reliability.

The driving criteria of this constructive process strategy are as follows:

3.1.1.1 Do It Right the First Time. This approach stresses the use of all means necessary to define and build a product with no faults. "Doing it right the first time" must stress error prevention and fault avoidance through the following practices:

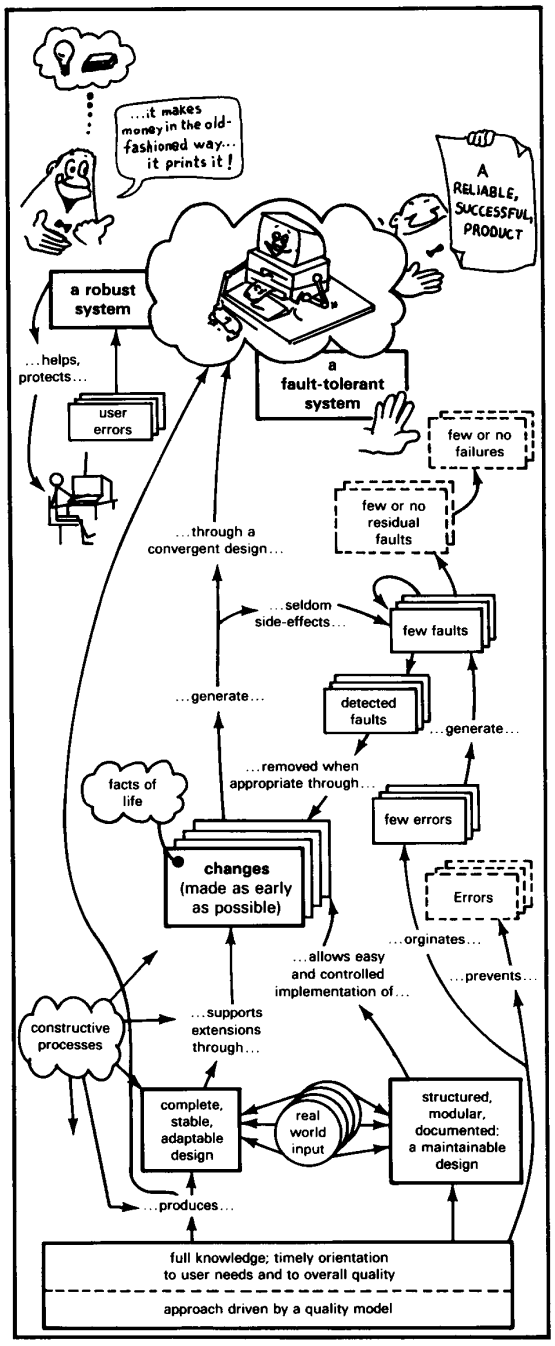


Fig 3.1.1-2
Constructors of a Reliable Product

(1) Employment of skilled, competent personnel is of foremost importance in building reliable products. It is important in all phases of development, especially the identification of user requirements and needs in the concept phase. The strategy stresses the necessity of intensely studying the stated and unstated needs of the user application, and of having sufficient experience in solving similar problems to apply to the present problem. For this reason, the concept phase is considered a major life cycle component toward a complete and stable definition of the objectives of a project. Quality work requires extensive knowledge of the application, which is often forgotten in the intermediate design phases, and remembered only when the user realizes some important features are missing during operation and maintenance.

(2) Early user involvement in the conception of the product through techniques such as prototyping must be stressed to ensure that usability matches failure-freeness in importance.

(3) Modern methods, languages and automated tools should be selected to promote productivity and quality in the activities of specification, design, coding and configuration management. The selected languages and tools must also support the verification activities through automation of requirements tracing, completeness and consistency analysis, performance simulation, complexity measurements, and configuration definition and control. Automation significantly reduces the risk of human errors.

3.1.1.2 Detect It Early; Fix It as Soon as Practicable. Detection and repair should be properly planned activities. Verification [1] is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. Validation [1] is the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Early detection of faults throughout the life cycle through reviews, inspections, prototyping, simulation, or proof of correctness of the product will improve reliability. During the operations and maintenance phase, faults do not necessarily have to be fixed immediately. The severity of the problem, the frequency of occurrence of the problem, and the opportunity to manage and test the new configuration should weigh in the decision of when to fix a fault.

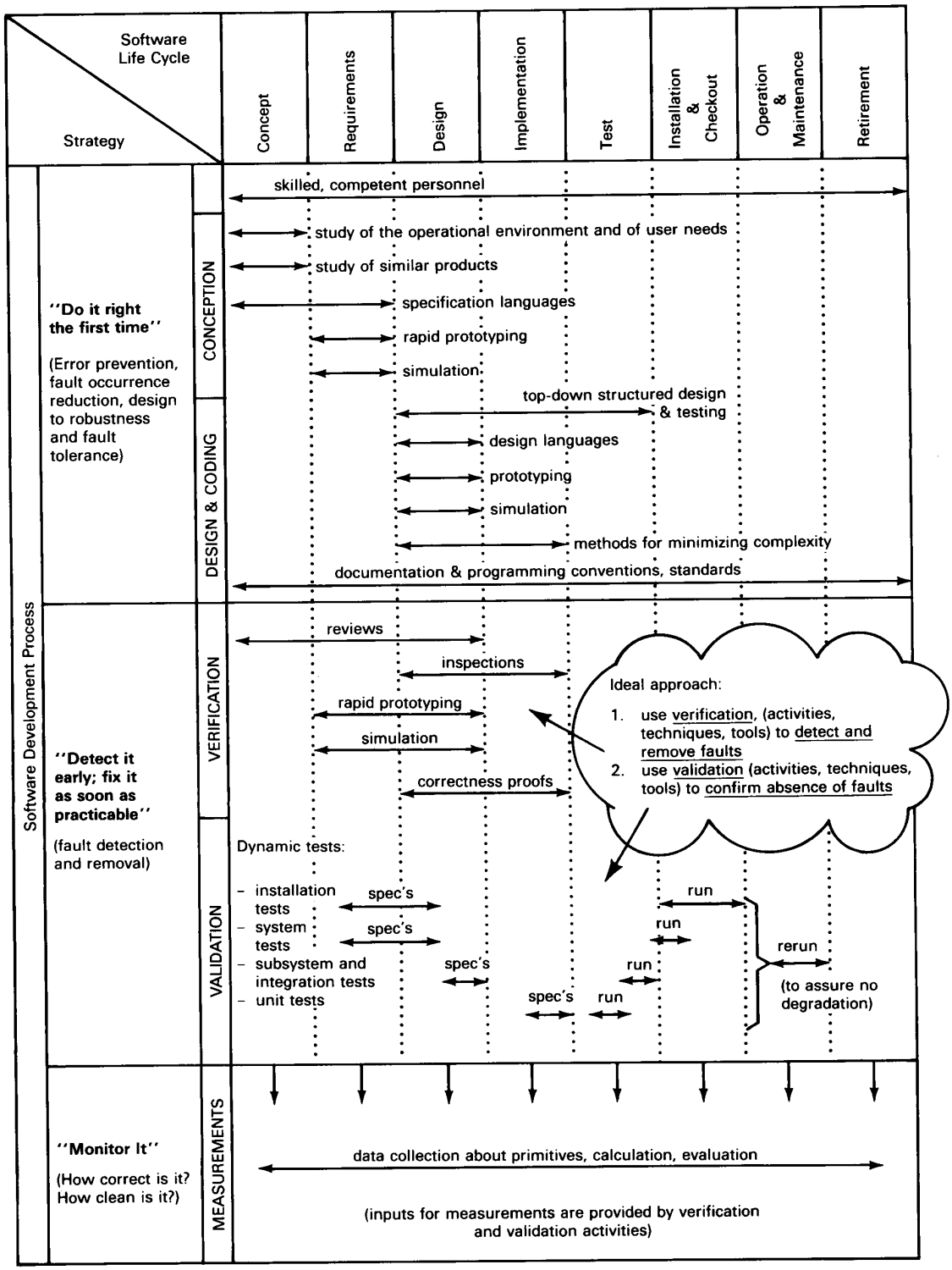


Fig. 3.1.1-3
Strategy of the Constructive Approach for Development of Reliable Software

Timely design of software tests, plans, and procedures is necessary to determine required resources and schedule. Early design of tests contributes to earlier detection of consistency and completeness faults, and ensures the testability of the requirements.

Reliability cannot be tested into a product. More testing does not necessarily mean better reliability. However, dynamic tests, the main tool for validation, have to be used cost-effectively with measures to provide the widest test coverage of a product. In the light of a constructive approach to reliable software, dynamic tests should be viewed as a tool for confirming the absence of faults rather than as the preferred tool for detecting them. When system test design is driven by user requirements, and when test samples are randomly selected based on defined probabilities of product inputs, testing contributes effectively to this constructive approach by providing an independent assessment of the software execution in representative operating environments. The costs of early verification efforts can be offset by savings from less repetition of dynamic tests, because most faults should be detected during verification.

3.1.1.3 Monitor It. Primitive data should be collected and evaluated throughout the life cycle.

Inegration between measurements and verification and validation activities is of paramount importance. There cannot be good measurements without a well-organized verification and validation process that provides the input data for measurements.

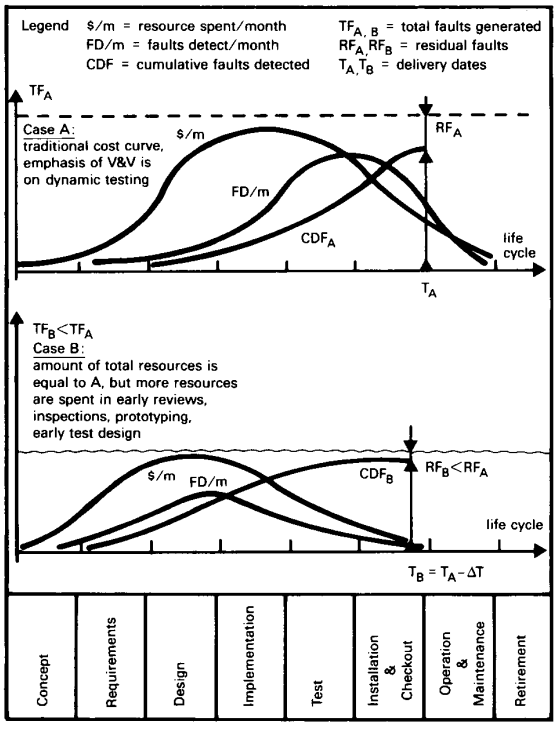
Measurements evaluation is used not only for software quality evaluation, but also for fault detection. Each instance where the criterion used for evaluation indicates a "below threshold" value should be considered an instance of a software fault. For example, measures of processing times higher than a given value may be sufficient reason to reject the implemented code and to redesign it.

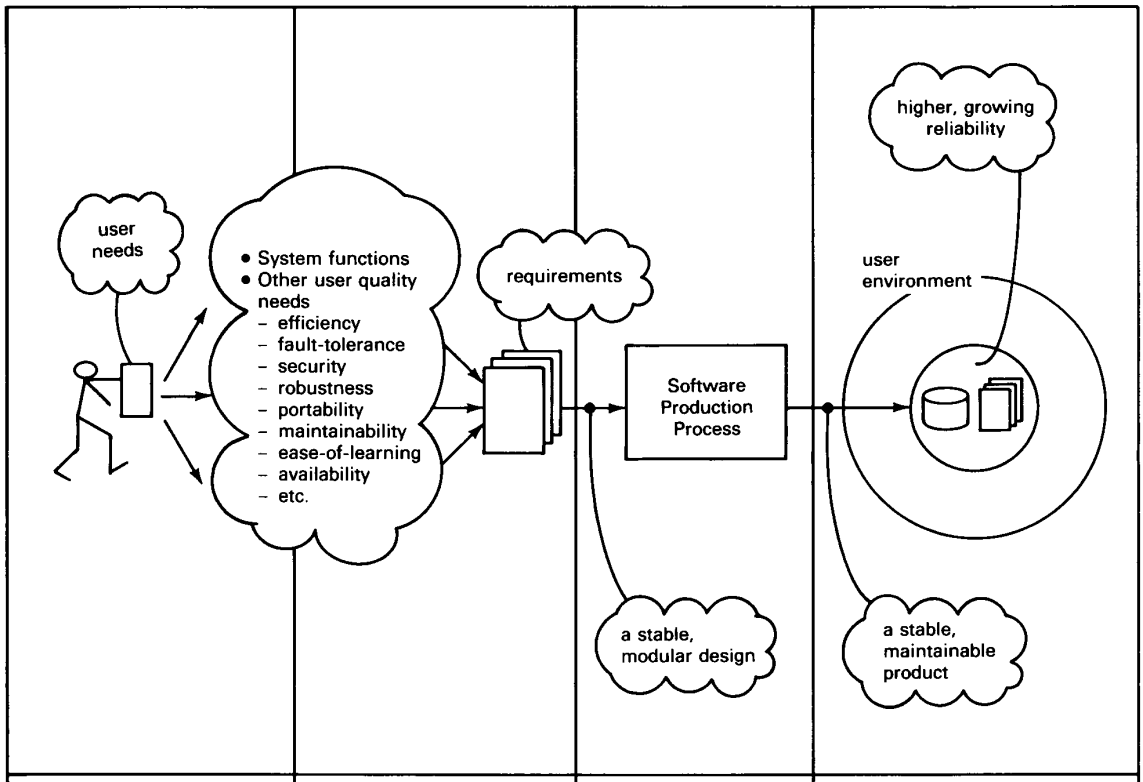
3.1.2 Reliability Optimization within Project Constraints. Software reliability should not be considered separately from other major factors: resources (cost), performance, and schedules. Software reliability should be optimized with respect to the project constraints. For example, if reliability and cost are considered as a ratio in which to attempt to increase the reliability and minimize the cost, then the project should seek the more efficient fault detection and removal activities. If reliability and schedule are jointly considered, early fault detection should be emphasized. Each project should assign priorities to these project factors and seek to optimize reliability within these constraints.

The cost of fixing faults rises significantly from the time they are generated to the time they are detected. It is more efficient to dedicate resources to fault detection and removal as soon after fault insertion as possible. Total cost comparisons for various levels of verification and validation efforts have shown that the savings of a verification and validation program are substantial. Figure 3.1.2-1 illustrates these considerations. Cases A and B have an equal cumulative amount of costs but different distribution of costs in time. In Case B the verification program puts greater emphasis, hence more resources, in early reviews, inspections, prototyping, and dynamic test design than in Case A.

3.1.3 Software Reliability in the Context of Product Quality. An analysis of the relationship of software reliability to other quality objectives is provided. The purpose of this discussion is to clarify the relationship among selected measures, software reliability and other quality objectives in general.

Fig 3.1.2-1
Different Approaches in Verification and Validation Programs





*During the operational phase, the quality model for software reliability is a function of product stability and adaptability.

Fig 3.1.3.1-1
Quality Model for Software Reliability

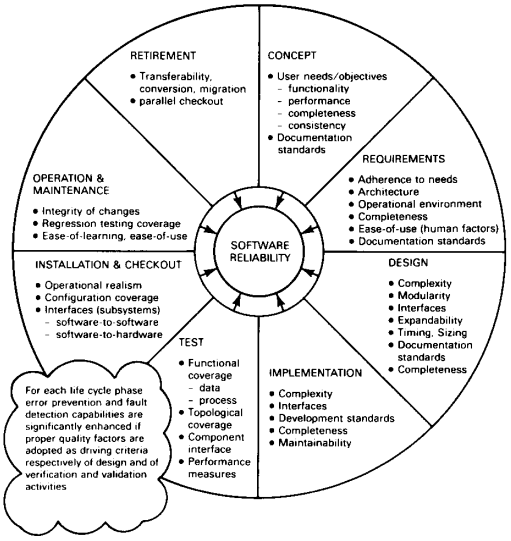


Fig 3.1.3.2-1
Quality Factors Impacting Software Reliability

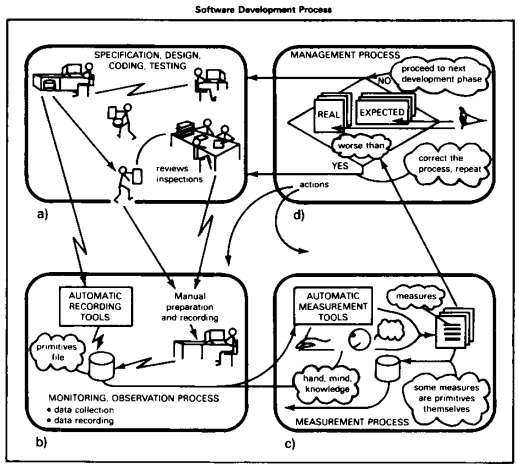


Fig 3.2-1
Measurement Information Flow: An Integral Part of Development Process

and goals. Other primitives, produced by verification and validation activities, include errors, faults and failures.

A two-step process is recommended to prepare for data collection:

(1) For each potential measure selected from IEEE Std 982.1-1988 [2], determine the data needed to perform the measurement from the list of primitives.

(2) Develop project practices for organizing data so information related to events during the development effort can be stored in a data base and retained for historical purposes and analysis. Automated tools integrated with the development environment should be considered.

Patient, accurate data collection and recording is the key to producing accurate measurements. If not well-organized, data collection can become very expensive. To contain these costs, the information gathering should be integrated with verification and validation activities. Appropriate mechanisms, such as forms or direct terminal entry, should be used to record the results of reviews. Use of automated tools for data retrieval and data entry can also reduce costs. Properly augmented configuration change control systems may furnish an ideal support environment for recording and evaluating most of the data for reliability assessment.

An important outgrowth of implementing a measurement effort is the ability to evaluate the results of collected data, develop measures, and apply analysis of the results to the end product. By tracking progress and evaluating the results, the measurement effort can be used to assess product reliability growth as well as the effectiveness of the development and support processes.

3.3 Measurement Selection Criteria. Current technology in software reliability is limited largely to the use of models that rely exclusively on test data applied to software at the end of development. The measures selected for the standard were extracted from available literature and from the collective experience of practitioners in the field. The primary emphasis was to identify a varied set of measures that cover the many aspects of the concept of reliability. This led to the inclusion of faults and failures as well as models for the traditional reliability calculations.

The constructive approach expanded the search to measures that promote reliability by minimizing residual faults through error prevention and early fault detection and removal. Therefore, the standard includes product measures to determine how carefully a product was verified (eg, completeness and traceability) and complexity measures to accommodate the possibility of change to the system.

Careful consideraton was given to the selection of process measures that monitor the quality of the development process and identify the need to make changes early in the process. Some process measures monitor the accuracy and completeness of development and test activities through examination of product properties (such as completeness and consistency). Other process measures monitor the efficiency of the activities (eg, error distribution). The process measures have the potential to stimulate adoption of good engineering practices at the proper project phase, specifically encouraging early fault detection.

A key consideration in the selection of the measures was to address the full range of the life cycle, a varied audience of both users and software and systems engineers, and a range of audience skills in reliability measurement.

The user class, which may include the owner, the operator and the recipient of the system capability, is more interested in measures related to system availability and operability, such as mean time to failure. The software engineer is more interested in software faults that cause excessive failure rates.

Reliability measures take on both aspects — system failures and software faults. The user can focus on system availability and capability; whereas the engineer can focus on the fault in the software product. Other measures, such as complexity, are of interest to managers and developers, assisting them to improve processes and control costs, schedules, personal productivity, and professional skills.

From the point of view of skill required, the measures are arranged in increasing order of difficulty. This allows individuals with little knowledge of computational methodologies to start making meaningful measurements. Subsequent measures depend upon increasing experience and sophistication with computational methodologies and automated assists.

4. Measure Organization and Classification

The measures in this guide are organized by the two basic objectives of reliability. The first objective is product maturity assessment — a certification of product readiness for operation (including an estimation of support necessary for corrective maintenance). The second objective of reliability is process maturity assessment — an ongoing evaluation of a software factory's capability to produce products of sufficient quality for user needs. Process maturity assessement includes product maturity assessment with the objective of process repair when necessary.

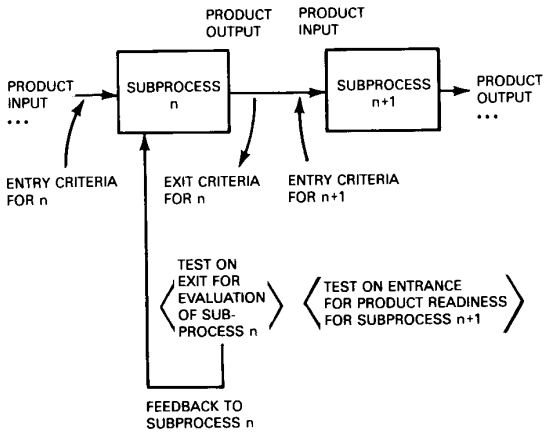
With hardware factories, process maturity assessment is usually performed through the technique of statistical process control. In software, this technique has been hampered by the lack of clearly defined measurables to relate product quality deficiencies to process cause. The measures in this guide provide the means to apply statistical process control to software.

The software life cycle can be modeled and refined into a sequence of subprocesses. Each subprocess has inputs that should meet well-defined entry criteria and outputs that should meet well-defined exit criteria.

An assessment of the product at the completion of a subprocess (the output against the exit criteria) not only evaluates the readiness of that individual product to enter the next subprocess, it serves as an indicator of the quality of the subprocess itself. It should be noted that failure of a product to pass the established exit criteria is a signal for further analysis. Either the subprocess could be defective or the criteria could be too stringent. Corrective action should be taken only after sufficient analysis (Fig 4-1).

For each measure the following topics are detailed in the Appendix: implementation, interpretation, considerations, training, an example, benefits, experience, and references.

Fig 4-1
Generalized Process Control



NOTE: Entry criteria n+1 might differ from exit criteria n since owners of subprocess n, entry criteria n, and exit criteria n could be separate organizations from those of n+1.

These measures can be divided into two functional categories: product and process. Product measures are applied to the software objects produced and are divided into six subcategories. Process measures are applied to the life cycle activities of development, test and maintenance and are divided into three subcategories.

The functional categories are as follows:

Product Measures

- Errors, faults, and failures
- Mean time to failure; failure rate
- Reliability growth and projection
- Remaining product faults
- Completeness and consistency
- Complexity

Process Measures

- Management control
- Coverage
- Risk, benefit, cost evaluation

In addition to the functional classification, the measures are also classified by the software life cycle phases in which their use is appropriate. Typical software life cycle phases based upon IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology [1] are used to illustrate when each of the measures may be applied. The life cycle phases are as follows:

- Concepts
- Requirements
- Design
- Implementation
- Test
- Installation and Checkout
- Operation and Maintenance
- Retirement

Finally, the measures can be classified as indicators or predictors.

4.1 Functional Classification. The measure classification matrix (Table 4.1-1) is a cross index of measures and functional categories. It can be used to select measures applicable to each category. For each functional category, a matrix is presented that cross-references the measures with the phases of the software life cycle in which they are applicable (Tables 4.2-1 through 4.2-9).

Table 4.1-1 includes an experience rating of 0, 1, 2, or 3. A "0" indicates that the measure has been formalized but has not been sufficiently validated to have operational experience. A "1" means that the measure has had limited use in the software community. This could be because the measure has been used only within an organization and has not had industry-wide exposure, or because the measure is not sufficiently well known to have been used on many projects. A "2" means that the measure has had moderate experience and a "3" means that it has had extensive experience. In no way does the experience rating imply that one measure is better than another.

Table 4-1
Category: Errors, Faults, Failures Counting

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
1. Fault density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
2. Defect density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
3. Cumulative failure profile	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
7. Requirements traceability		Δ	Δ				Δ	
8. Defect indices	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	

Table 4.1-1
Measure Classification Matrix

	Product Measures						Process Measures		
Measures (Experience)	Errors, Faults, Failures	Mean Time to Failure; Failure Rate	Reliability Growth & Projection	Remaining Product Faults	Completeness & Consistency	Complexity	Management Control	Coverage	Risk, Benefit, Cost Evaluation
1. Fault density (2)	X								
2. Defect density (3)	X								
3. Cumulative failure profile (1)	X								
4. Fault-days number (0)	X						X		
5. Functional or modular test coverage (1)					X			X	X
6. Cause and effect graphing (2)					X			X	
7. Requirements traceability (3)	X				X			X	
8. Defect indices (1)	X						X		
9. Error distribution(s) (1)							X		
10. Software maturity index (1)			X						X
11. Man hours per major defect detected (2)							X		X
12. Number of conflicting requirements (2)	X				X			X	
13. Number of entries/exits per module (1)					X	X			
14. Software science measures (3)				X		X			
15. Graph-theoretic complexity for architecture (1)						X			
16. Cyclomatic complexity (3)					X	X			
17. Minimal unit test case determination (2)					X	X			
18. Run reliability (2)			X						
19. Design structure (1)						X			
20. Mean time to discover the next K faults (3)									X
21. Software purity level (1)			X						
22. Estimated number of faults remaining (seeding) (2)				X					
23. Requirements compliance (1)	X				X			X	
24. Test coverage (2)					X			X	
25. Data or information flow complexity (1)						X			
26. Reliability growth function (2)			X						
27. Residual fault count (1)				X					
28. Failure analysis using elapsed time (3)			X	X					
29. Testing sufficiency (0)			X					X	
30. Mean-time-to-failure (3)		X	X						
31. Failure rate (3)		X							
32. Software documentation & source listings (2)					X				
33. RELY - (Required Software Reliability) (1)								X	X
34. Software release readiness (0)									X
35. Completeness (2)					X				
36. Test accuracy (1)				X	X			X	
37. System performance reliability (2)			X						
38. Independent process reliability (0)			X						
39. Combined HW/SW system operational availability (0)			X						

4.1.1 Product Measures. The product measures address cause and effect of the static/dynamic aspects of both projected reliability prior to operation, and operational reliability. As an example, reliability may change radically during the maintenance effort, due to the complexity of the system design. The six product measure subcategories address these dimensions of reliability.

(1) Errors, faults, failures—Count of defects with respect to human cause, program bugs, and observed system malfunctions.

(2) Mean-time-to-failure; failure rate—Derivative measures of defect occurrence and time.

(3) Reliability growth and projection—The assessment of change in failure-freeness of the product under testing and in operation.

(4) Remaining products faults—The assessment of fault-freeness of the product in development, test, or maintenance.

(5) Completeness and consistency—The assessment of the presence and agreement of all necessary software system parts.

(6) Complexity—The assessment of complicating factors in a system.

4.1.2 Process Measures. The process measures address cause and effect of both the static and dynamic aspects of the development and support management processes necessary for maximizing quality and productivity. The three process measure subcategories address the process dimension of reliability.

(1) Management control measures address the quantity and distribution of error and faults and the trend of cost necessary for defect removal.

(2) Coverage measures allow one to monitor the ability of developers and managers to guarantee the required completness in all the activities of the life-cycle and support the definition of corrective actions to carry the coverage indicators to the desired levels.

(3) Risks, benefits, and cost evaluation measures support delivery decisions based both on technical and cost criteria. Risk can be assessed based on residual faults present in the product at delivery and the cost associated with the resulting support activity.

Table 4.2-1
Life Cycle Classification Matrix: Errors, Faults, Failures Counting

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
1. Fault density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
2. Defect density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
3. Cumulative failure profile	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
7. Requirements traceability		Δ	Δ				Δ	
8. Defect indices	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	

Table 4.2-2
Life Cycle Classification Matrix: Mean-Time-to-Failure; Failure Rate

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
30. Mean time to failure					Δ	Δ	Δ	
31. Failure rate					Δ	Δ	Δ	

Table 4.2-3
Life Cycle Classification Matrix: Reliability Growth and Projection

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
10. Software maturity index	Δ	Δ	Δ			Δ	Δ	
18. Run reliability					Δ	Δ	Δ	
21. Software purity level					Δ	Δ	Δ	
26. Reliability growth function					Δ	Δ	Δ	
28. Failure analysis using elapsed time					Δ	Δ	Δ	
29. Testing sufficiency					Δ	Δ		
30. Mean time to failure					Δ	Δ	Δ	
37. System performance reliability		Δ	Δ	Δ	Δ	Δ	Δ	
38. Independent process reliability						Δ	Δ	
39. Combined HW/SW operational availability					Δ	Δ	Δ	

Table 4.2-4
Life Cycle Classification Matrix: Remaining Faults Estimation

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
14. Software science measures				Δ			Δ*	
22. Estimated number of faults remaining (seeding)					Δ	Δ	Δ	
27. Residual fault count					Δ	Δ	Δ	
28. Failure analysis using elapsed time					Δ	Δ	Δ	
36. Test accuracy					Δ			

*If source code is changed

Table 4.2-5
Life Cycle Classification Matrix: Completeness, Consistency

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
6. Cause and effect graphing		Δ	Δ	Δ	Δ		Δ	
7. Requirements traceability		Δ	Δ				Δ	
12. Number of conflicting requirements		Δ					Δ	
16. Cyclomatic complexity (testing phase)				Δ	Δ		Δ	
23. Requirements compliance		Δ					Δ	
24. Test coverage		Δ	Δ		Δ	Δ	Δ	
32. Software documentation and source listings				Δ	Δ	Δ	Δ	
35. Completeness		Δ	Δ				Δ	
36. Test accuracy					Δ			

Table 4.2-6
Life Cycle Classification Matrix: Complexity

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout	Operation & Maintenance	Retirement
13. Number of entries/exits per module			Δ	Δ			Δ	
14. Software science measures				Δ			Δ	
15. Graph-theoretic complexity for architecture		Δ	Δ				Δ	
16. Cyclomatic complexity			Δ	Δ			Δ	
17. Minimal test case determination				Δ	Δ		Δ	
19. Design structure			Δ				Δ	
25. Data or information flow complexity			Δ	Δ			Δ	

Table 4.2-7
Life Cycle Classification Matrix: Management Control

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout	Operation & Maintenance	Retirement
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
8. Defect indices		Δ	Δ	Δ	Δ	Δ	Δ	
9. Error distribution(s)		Δ	Δ	Δ	Δ	Δ	Δ	
11. Man-hours per major defect detected		Δ	Δ	Δ	Δ	Δ	Δ	
20. Mean time to discover next *K* faults					Δ	Δ	Δ	

Table 4.2-8
Life Cycle Classification Matrix: Coverage

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
6. Cause and effect graphing		Δ	Δ	Δ	Δ		Δ	
7. Requirements traceability		Δ	Δ				Δ	
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	
24. Test coverage		Δ	Δ	Δ	Δ	Δ	Δ	
29. Testing sufficiency					Δ		Δ	
32. Software documentation & source listings				Δ	Δ	Δ	Δ	
33. RELY - Required Software Reliability	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
35. Completeness		Δ	Δ				Δ	
36. Test accuracy					Δ			

Table 4.2-9
Life Cycle Classification Matrix: Risk, Benefit, and Cost Evaluation

	Life Cycle Phase							
Measures	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
10. Software maturity index	Δ	Δ	Δ			Δ	Δ	
11. Man-hours per major defect detected		Δ	Δ	Δ	Δ	Δ	Δ	
14. Software science measures				Δ			Δ	
18. Run reliability					Δ	Δ	Δ	
19. Design structure			Δ					
20. Mean time to discover the next K faults					Δ	Δ	Δ	
21. Software purity level					Δ	Δ	Δ	
27. Residual fault count					Δ	Δ	Δ	
31. Failure rate					Δ	Δ	Δ	
33. RELY - Required Software Reliability	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
34. Software release readiness					Δ	Δ		

4.2 Life Cycle Classification. To assist the user in more practical considerations, such as "When are the measures applied?" the life cycle classification (Fig 4.2-1) has been developed. The life cycle has been divided into early, middle, and late segments. In the early segment, the measures relate to the potential causes of system reliability. The middle segment measures relate to the reduction of process errors that can improve the efficiency of software development. The late segment measures relate to actual system performance reliability.

For each life cycle phase a set of possible measures that might be applied during the development effort should be identified. Selection need not be limited to one measure per development phase, but on the other hand, it is expected that overzealous selection of measures would not benefit a project.

4.2.1 Early Life Cycle Segment. The early segment — concepts, requirements, and design — is the period when many of the characteristics of the system, including potential reliability, are determined. Problems in these early phases tend to result from premature constraints on the system design that limit system solution alternatives and affect the real system objectives, which include reliability. Emphasis and focus of the measures in the early stages are on the goals and objectives of the system. The concrete feedback they provide aids in achieving completeness and consistency in the definition of user requirements as well as the design.

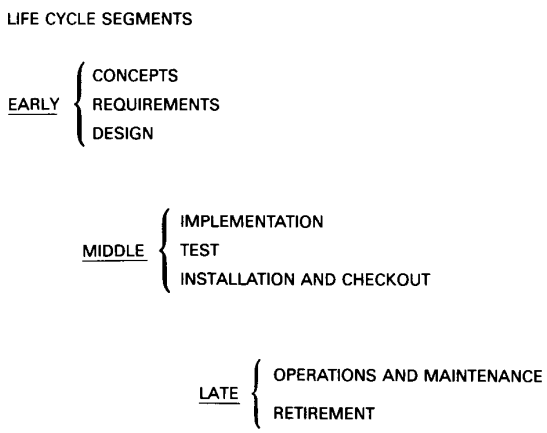
4.2.2 Middle Life Cycle Segment. The middle segment of the software life cycle includes implementation, test, installation and checkout. The software engineer's goal in this segment is to develop fault-free software efficiently. The objectives of reliability measurement in this segment include not only reliability projection, but also guidance of the process so that fault-free software is developed efficiently. For reliability projection, faults found in this segment can relate directly to operational failures, expecially when development activity is halted prematurely. These faults, in fact, can be used to determine software readiness. For the process evolution objective, these faults can be considered process errors and can be directly related to the effectiveness and efficiency of the previously applied processes. Cause-analysis can relate the fault to that point in the process where the error was made. Proper process-control techniques applied to these measures can indicate if and when action should be taken to change the process.

4.2.3 Late Life Cycle Segment. The late segment of the software life cycle — operation, maintenance, and retirement — is where system performance, as an expression of reliability and availability, can be directly measured using the traditional software reliability measures. Current software product performance can be used to project future performance. In addition, future development can be improved by relating the problems in the current software product to the steps in the development process that produced them.

In this segment there are two audiences for software reliability measures — the user and the software engineer. The users are the owner, the operator, and the recipient of the end product of the system capability. The software engineers include both the original developer and the subsequent maintainer. In most cases, the user is more interested in system availability and operability; the software engineer is more interested in the origins of software faults that cause failures. The user focuses on the capability of the system; the software engineer focuses on the faults in the restricted software domain.

Reliability measures should address both system failures and software faults to provide useful information to both user and software engineer. There are traditional measures of software performance that provide both a user and a software engineer view of system reliability. Many of these measures — for example, mean-time-to-failure (user) and software fault discovery rate (software engineer) — serve as primitives for reliability projections based on modeling techniques.

Fig 4.2-1
Life Cycle Classification



4.3 Indicators and Predictors. Measures can also be organized with respect to their primary function as indicators or predictors. Indicator measures provide an assessment of reliability at the point in time the measurement is taken. Predictor measures project forecasted behavior as suggested by past behavior. Historical data based upon consistently applied indicator measures are the basic input for predictor measures. The higher the integrity and quantity of the historical record, the better results can be expected from the predictor measures. Note that some measures are both predictor and indicator in nature.

Indicator measures can be used to determine completion of activities throughout the life cycle. In this context, established goals are considered successfully achieved based on the value of the indicator measure. Using the history of indicator measurements, future behavior can be determined based upon predictor measures.

Table 4.3-1 provides a cross reference of measures—first, to the object or events that are subjects of measurement, and second, to the primary indicator or predictor nature of the measure.

Table 4.3-1
Indicator/Predictor Measure Classification

Object/Event	Measure
Requirements	
Indicator	4.6, 4.7, 4.12, 4.23, 4.33, 4.35
Predictor	—
Design	
Indicator	4.13, 4.15, 4.19, 4.25
Predictor	—
Source code	
Indicator	4.16, 4.17
Predictor	4.14
Test set	
Indicator	4.5, 4.24
Predictor	—
Any product	
Indicator	4.32
Predictor	—
Software faults	
Indicator	4.1, 4.2, 4.4, 4.8, 4.9, 4.11
Predictor	4.22, 4.29, 4.36
Software failures	
Indicator	4.3
Predictor	4.18, 4.20, 4.21, 4.26, 4.27, 4.28, 4.30, 4.31, 4.39
Release package	
Indicator	4.10, 4.34
Predictor	—
Performing object code	
Indicator	4.37
Predictor	—
Tested object modules	
Indicator	4.38
Predictor	—

5. Framework for Measures

Software reliability measurements take place in an environment that includes user needs and requirements, a process for developing products meeting those needs, and user environment within which the delivered software satisfies those needs. This measurement environment establishes a framework for determining and interpreting indicators of software reliability.

5.1 Measurement Process. In this section, a measurement process is detailed that can constructively affect the delivered reliability of software through the selection and interpretation of indicators of software reliability. This process formalizes the data collection practices in both development and support. It provides for product evaluation at major milestones in the life cycle. Furthermore, it relates measures from one life cycle phase to another. In summary, this process provides the basis for reliability measurement of a product.

5.2 Stages of a Measurement Process. The software reliability measurement process can be described in nine stages (Fig 5.2-1). These stages may overlap or occur in different sequences depending on organization needs. These stages provide a useful model for describing measurement process issues. Each of these stages in the measurement process influences the production of a delivered product with the potential for high reliability. Other factors influencing the measurement process include the following: a firm management commitment to continually assess product and process maturity, or stability, or both during the project; use of trained personnel in applying the measures to the project in a useful way; software support tools; and a clear understanding of the distinctions among errors, faults, and failures.

5.2.1 Stage 1: Plan Organizational Strategy. Initiate a planning process. Form a planning group and review reliability constraints and objectives, giving consideration to user needs and requirements. Identify the reliability characteristics of a software product necessary to achieve

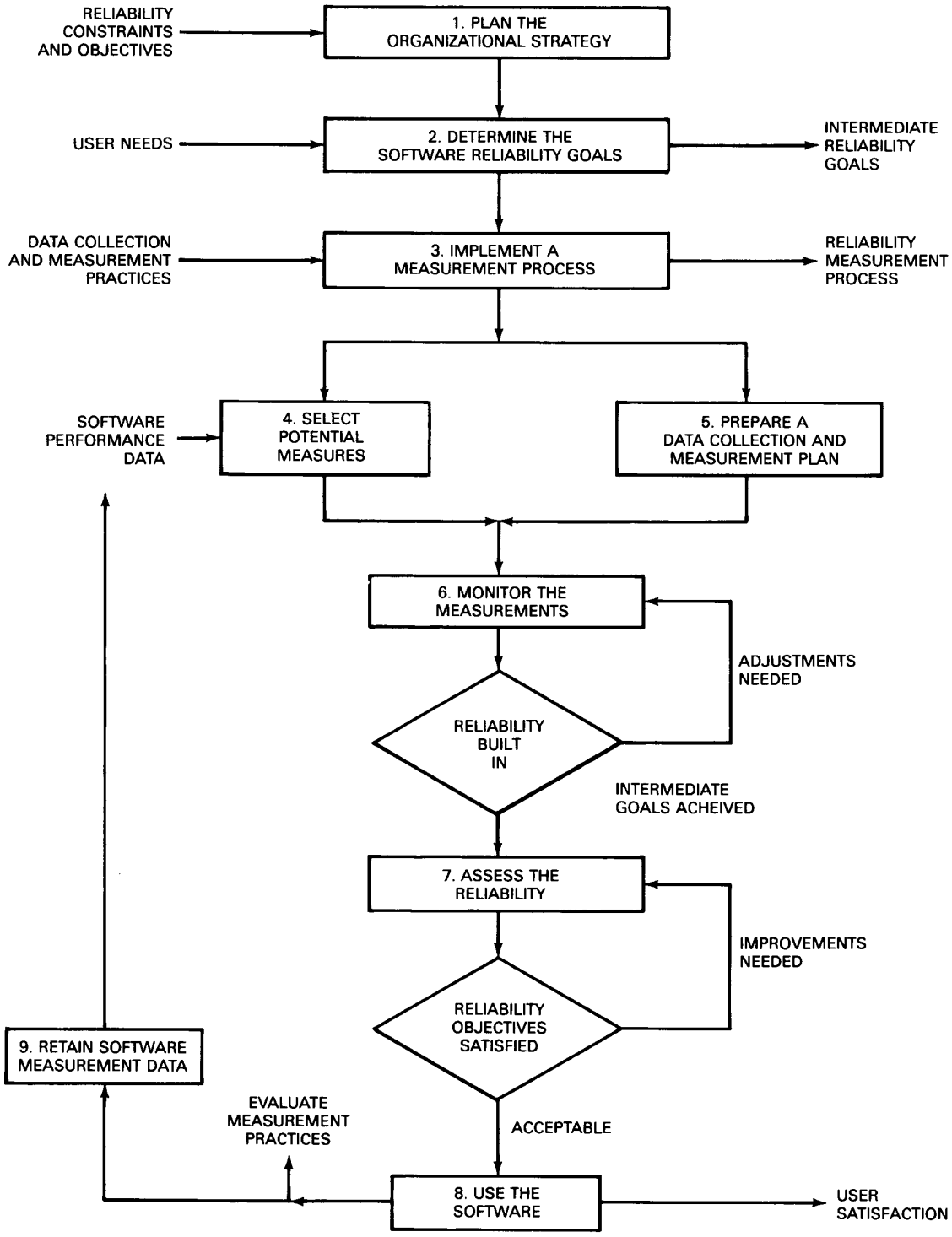


Fig 5.2-1
Reliability Measurement Process

these objectives. Establish a strategy for measuring and managing software reliability. Document practices for conducting measurements.

5.2.2 Stage 2: Determine Software Reliability Goals. Define the reliability goals for the software being developed in order to optimize reliability in light of realistic assessments of project constraints, including size, scope, cost, and schedule.

Review the requirements for the specific development effort, in order to determine the desired characteristics of the delivered software. For each characteristic, identify specific reliability goals that can be demonstrated by the software or measured against a particular value or condition. Establish an acceptable range of values. Consideration should be given to user needs and requirements, including user perception of software reliability and the system environment in which the software must interact.

Establish intermediate reliability goals at various points in the development effort to assist in achieving the reliability goals.

5.2.3 Stage 3: Implement Measurement Process. Establish a software reliability measurement process that best fits an organization's needs.

Review the process described in stages 4 through 8 and select those stages that best lead to optimum reliability. Add to or enhance these stages as needed to reflect the specific work environment. Consider the following suggestions when establishing the measurement process:

(1) Select appropriate data collection and measurement practices designed to optimize software reliability.

(2) Document the measures required, the intermediate and final milestones when measurements are taken, the data collection requirements, and the acceptable values for each measure.

(3) Assign responsibilities for performing and monitoring measurements, and provide necessary support for these activities from across the internal organization.

(4) Initiate a measure selection and evaluation process.

(5) Prepare educational materials for training personnel in concepts, principles, and practices of software reliability and reliability measures.

After selecting and establishing the steps of the measurement process, reduce it to an organization operation procedure. Maintain complete records of the measurement process to provide a historical perspective so that future reliability measurement programs may benefit.

5.2.4 Stage 4: Select Potential Measures. Identify potential measures that would be helpful in achieving the reliability goals established in Stage 2.

Once the organization software reliability goals are established, select the measures best suited to the development and support environments.

Use the classification scheme described in Section 4 to determine which category or categories of reliability measurements would be most useful to the project. When possible, measures should be chosen within a particular category to show a continuous relationship between early indicators and the final software reliability achieved at delivery.

Measures should be selected to span categories and development phases. Selection need not be limited to one measure per category or phase. However, overzealous selection of measures should be avoided.

The classification scheme assists in identifying and applying related measures across a development project (commonality) or in applying the same measure(s) to several projects (comparison).

5.2.5 Stage 5: Prepare Data Collection and Measurement Plan. Prepare a data collection and measurement plan for the development and support effort. For each potential measure, determine the primitives needed to perform the measurement. Data should be organized so that information related to events during the development effort can be properly recorded in a data base and retained for historical purposes.

For each intermediate reliability goal identified in Stage 2, identify the measures needed to achieve the goal. Identify the points during development when the measurements are to be taken. Establish acceptable values or a range of values to assess whether the intermediate reliability goals are achieved.

Include in the plan an approach for monitoring the measurement effort itself. The responsibility for collecting and reporting data, verifying its accuracy, computing measures, and interpreting the results should be described. Use historical data where possible.

5.2.6 Stage 6: Monitor Measurements. Once the data collection and reporting begins, monitor the measurements and the progress made during development, so as to manage the reliability and thereby achieve the goals for the delivered product. The measurements assist in determining whether the intermediate reliability goals are

achieved and whether the final goal is achieveable. Analyze the measures and determine if the results are sufficient to satisfy the reliability goals. Decide whether a measure result assists in affirming the reliability of the product or process being measured. Take corrective action as necessary to improve the desired characteristics of the software being developed.

5.2.7 Stage 7: Assess Reliability. Analyze measurements to ensure that reliability of the delivered software satisfies the reliability objectives and that the reliability, as measured, is acceptable.

Identify assessment steps that are consistent with the reliability objectives documented in the data collection and measurement plan. Check the consistency of acceptance criteria and the sufficiency of tests to satisfactorily demonstrate that the reliability objectives have been achieved. Identify the organization responsible for determining final acceptance of the reliability of the software. Document the steps involved in assessing the reliability of the software.

Ensure that reliability improvements in the software have been made. Demonstrate satisfactory achievement of all reliability goals by reliability measurements and internal tests and by field reliability measurements during use.

5.2.8 Stage 8: Use Software. Assess the effectiveness of the measurement effort and perform necessary corrective action. Conduct a follow-up analysis of the measurement effort to evaluate reliability assessment and development practices, record lessons learned, and evaluate user satisfaction with the reliability of the software. Identify practices that are ineffective and requirements that indicate a need to develop and refine future measurement efforts.

5.2.9 Stage 9: Retain Software Measurement Data. Retain measurement data on the software throughout the development and operation phases for use by future projects. This data provides a baseline for reliability improvement and an opportunity to compare the same measures across completed projects. This information can assist in developing future guidelines and standards.

6. Errors, Faults, and Failures Analysis for Reliability Improvement

Reliability is a high-level indicator of the operational readiness of a system. Measurement for reliability need not be viewed as a passive score to be achieved, but rather as an active means of improving the processes used to build and maintain systems.

A key to reliability improvement is the maintenance of an accurate history of errors and faults associated with failure events. This section presents an introduction to the dynamics of errors, faults, and failures, with the goal of improving reliability by minimizing future failure events through error analysis and process control.

6.1 Dynamics of Errors, Faults, and Failures. Errors, faults, and failures can be viewed as a series of cause and effect relationships (Fig 6.1-1). The source of errors is either developers or users. Those developer errors resulting in undetected faults in the product directly affect reliability. User errors, usually misunderstandings, can affect perceived reliability.

Failures may occur when a fault is encountered (for example, in the code or the user documentation), or when a how-to-use error is made by the user. Faults exist in the product due to errors made by the designer or programmer during the development process. How-to-use errors are originated by either human factor faults within the product, or by the user misunderstanding the product application.

Lacks within the product exist when a desired feature or characteristic is incomplete or missing from the software. These lacks or weaknesses are a subclass of faults.

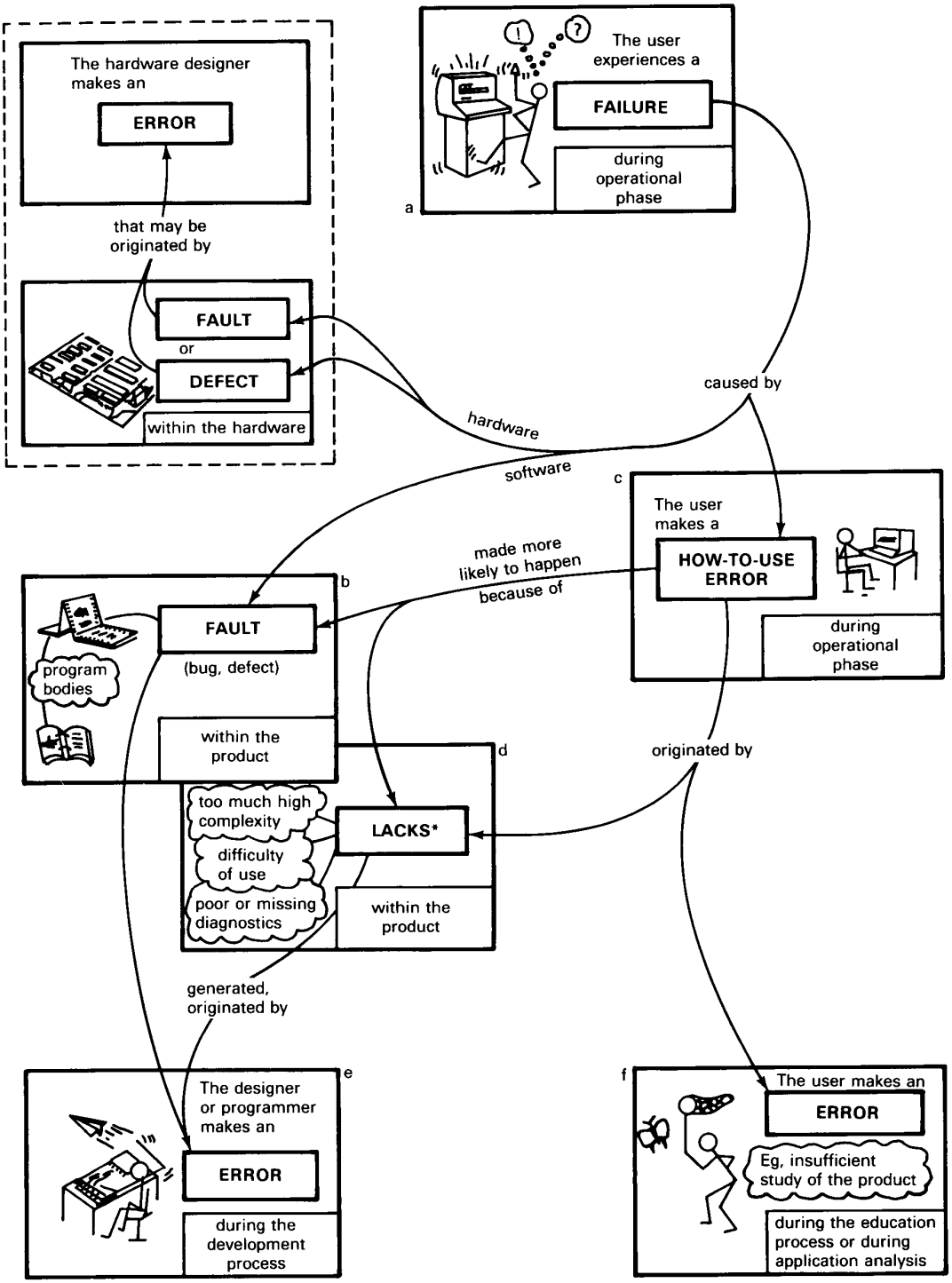
While features due to how-to-use errors may be caused by the interaction of the product and user, or by the actions of the user alone, they are often perceived by the average user as a failure of the product.

These interactions must be clearly understood in order to measure and control the development and support processes.

6.2 Analysis of Error, Fault, Failure Events

6.2.1 Failure Analysis. The purpose of failure analysis is to acquire knowledge of the most important "failure modes" of the product in the user environment. It is important to distinguish between failure due to product faults and failure due to user error. Failures due to user error may be caused by the following:

- lack of clear, well-structured user documentation
- complex user interfaces
- lack of training
- lack of support
- insufficient user educational or systems background



*Product lacks (or weaknesses) could also be seen as a subclass of faults.

Fig 6.1-1
The Error, Fault, Failure Chain

- insufficient user awareness of the operational environment or product purpose
- inconsistencies between product specifications and the user's application environment
- inadequate hardware configuration
- inadequate system performance.

An analysis of these failures can improve the human factors of the product based on a better understanding of the user's needs. The developer can improve documentation, application support, and system support. Further, the developer can suggest a tailored user assistance and training program. In order to prevent reoccurrence, the developer can monitor this failure type.

6.2.2 Fault Analysis. Failures due to developer or maintainer error are caused by product faults.

The purpose of fault analysis is to gather historical information on product faults in order to evaluate reliability, perform error analysis, sensitize defect removal processes, and identify fault-prone subsystems. This analysis is performed mainly with product measures.

6.2.3 Error Analysis. The purpose of error analysis is to gather historical information about error types in order to acquire insight into typical individual and organizational behavior in a software engineering environment. This information can be used to address human weaknesses in the development process.

Error analysis is usually performed with process measures. This analysis mainly aids in improvement of the review and testing activities.

6.3 Minimizing Failure Events. Reliability can be improved by minimizing future failure events. This requires that specific attention be given throughout the life cycle to fault avoidance, fault detection, and fault removal.

It is obvious that failures can be minimized by avoiding errors that generate the corresponding faults. Furthermore, once faults have been generated in development, it is more productive to strive for early detection and efficient removal. For faults discovered in support processes, it is important to determine the testing necessary to recertify the repaired system. In this case, prompt fault removal may not be as productive as periodic batched updates to the system.

6.4 Summary. Improved reliability, the end objective, can be achieved with an accurate history of errors, faults, and failures. An analysis of failures, and the faults and errors that cause them, provides a basis for improving the development and support processes. At the same time, this history provides a performance projection of future systems to be developed with these processes. The accuracy of this projection and the success of process improvements is directly related to the accuracy of the record of errors, faults, and failures. The measures in this document assume errors, faults, and failures as primitive objects or events to be counted.

Appendix

Measures for Reliable Software

(This Appendix is not a part of IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.)

A1. Fault Density

A1.1 Application. This measure can be used to perform the following functions:

(1) Predict remaining faults by comparison with expected fault density;

(2) Determine if sufficient testing has been completed, based on predetermined goals for severity class;

(3) Establish standard fault densities for comparison and prediction.

A1.2 Primitives. Establish the severity levels for failure designation.

F = Total number of unique faults found in a given time interval resulting in failures of a specified severity level

KSLOC = Number of source lines of executable code and non-executable data declarations in thousands

A1.3 Implementation. Establish severity, failure types and fault types.

(1) Failure types might include I/O (input, output, or both) and user. Fault types might result from design, coding, documentation, and initialization.

(2) Observe and log each failure.

(3) Determine the program fault(s) that caused the failure. Classify the faults by type. Additional faults may be found resulting in total faults being greater than the number of failures observed, or one fault may manifest itself by several failures. Thus, fault and failure density may both be measured.

(4) Determine total lines of executable and non-executable data declaration source code (KSLOC).

(5) Calculate the fault density for a given severity level as:

$F_d = F/K\text{SLOC}$

A1.4 Interpretation. Comparison of fault density measurements made on similar systems can give some indication of whether the current system has been sufficiently tested. Software reliability may be qualitatively assessed from this information. In earlier phases of the life cycle, when KSLOC is unavailable, some other normalizing factor (eg, thousands of words of prose) agreed upon for that life cycle phase could be used.

A1.5 Considerations

(1) Test results must be thoroughly analyzed to ensure that faults, not just failures, are identified. Test coverage must be comprehensive.

(2) KSLOC must be consistently calculated (executable code and data).

(3) It should be recognized when comparing products that expected fault densities and measured fault densities have class and severity.

A1.6 Training. Minimal training is required for the use of this measure. Previous software testing experience, however, is required to establish fault and failure classifications. Good record keeping techniques and a source counting tool are essential.

A1.7 Example

(1) Sample data collection form:

Failure/Fault Log Date: / /

Software Tested: ____ Version: ____ KSLOC: ____

Test Suite: ____ Recorder: ____

Hardware Configuration: ____

Software Configuration: ____

Failure #	Fault #	Date	Description	Severity
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

(2) Sample calculation:
Let the number of failures be 21, the number of unique faults from the 21 failures be $F = 29$, and the number of lines of source code be 6000. Then the fault density is

F_d = F/KSLOC = 29/6 = 4.8 faults/KSLOC.

A1.8 Benefits. This measure is simple to use and provides a first approximation for reliability estimates. Collected data can be used to calculate other measures.

A1.9 Experience. Many organizations are using this measure and have developed their own guidelines but have not published their results. See the references for published examples.

A1.10 References

[A1] BOWEN, J. B. A Survey of Standards and Proposed Metrics for Software Quality Metrics. *Computer*, 1979, 12 (8), pp 37-41.

[A2] SHOOMAN, M. L. *Software Engineering Design/Reliability/Management.* New York: McGraw Hill, 1983, pp 325-329.

A2. Defect Density

A2.1 Application. The defect density measure can be used after design and code inspections of new development or large block modifications. If the defect density is outside the norm after several inspections, it is an indication that the inspection process requires further scrutiny.

A2.2 Primitives. Establish severity levels for defect designation.

D_i = total number of unique defects detected during the ith design, or code inspection process, or the ith life cycle phase;

I = total number of inspections to date;

KSLOD = In the design phase, the number of source lines of design statements in thousands;

KSLOC = In the implementation phase, the number of source lines of executable code and non-executable data declarations in thousands.

A2.3 Implementation. Establish a classification scheme for severity and class of defect. For each inspection, record the product size and the total number of unique defects.

For example, in the design phase calculate the ratio

$$DD = \frac{\sum_{i=1}^{I} D_i}{K\text{SLOD}}$$

This measure assumes that a structured design language is used. However, if some other design methodology is used, then some other unit of defect density has to be developed to conform to the methodology in which the design is expressed.

A2.4 Interpretation. This measure has a degree of indeterminism. For example, a low value may indicate either a good process and a good product or it may indicate a bad process. If the value is low compared to similar past projects, the inspection process should be examined. If the inspection process is found to be adequate, it should then be concluded that the development process has resulted in a relatively defect-free product.

If the measurement value is too high compared to similar past projects, the development process should be examined to determine if poor programming practices have resulted in poor software products. In such a case, a breakdown of defects by type and severity should be made into defect categories such as design, logic, standards, interface, commentary, requirements, etc. The significance of the defect density can then be determined and management action taken. For example, if the defects are high due to requirements, it would be appropriate to halt development, at least in one area, until the requirements are corrected. If, on the other hand, the development process is judged to be adequate, then the inspection process is serving a valuable pretest function.

After significant experience with this measure, it may be appropriate to establish lower values as goals in order to ensure continued reliability growth of the software development process.

A2.5 Considerations. This measure assumes a knowledge of the design and code inspection process. It further assumes consistent and standard data collection and recording. The inspection process should prove effective if strongly committed to and supported by management. The earlier the inspection data can be derived and assessed, the greater the benefit.

A2.6 Training. Although little or no training is needed for calculation of the measure, training in the inspection process is required.

A2.7 Example. Suppose, in the design phase,

I = 7

KSLOD = 8

$$\sum_{i=1}^{7} D_i = 78 \text{ (total defects found)}$$

Then,

$$DD = \frac{78}{8} = 9.8 \text{ (estimated defect density)}$$

A2.8 Benefits. The measure is easy to compute and it serves as one of the parameters to evaluate the effectiveness of the inspection process. Deviations from the norm serve as an indicator that immediate management attention is required, and therefore gives the developer a chance to make whatever corrections are necessary to maintain significant product control, reliability, and correctness.

A2.9 Experience. The analysis of this measure in several studies [A3, A4] has shown the utilization of the inspection process and the measurement of the collected data to be a highly effective management tool for software development and defect removal. In addition, the results have been highly consistent and the impact on error profiles during test has been seen.

A2.10 References

[A3] DOBBINS, J., and BUCK, R. Software Quality in the 80's. *Trends and Applications Proceedings*, IEEE Computer Society, 1981.

[A4] DOBBINS, J., and BUCK, R. Software Quality Assurance. *Concepts, Journal of the Defense Systems Management College*, Autumn, 1982.

[A5] FAGAN, MICHAEL E. Design and Code Inspection to Reduce Errors in Program Development. *IBM Systems Journal*, vol 15, no 3, Jul 1976, pp 102-211.

A3. Cumulative Failure Profile

A3.1 Applications. This is a graphical method used to:

(1) Predict reliability through the use of failure profiles;
(2) Estimate additional testing time to reach an acceptably reliable system;
(3) Identify modules and subsystems that require additional testing.

A3.2 Primitives. Establish the severity levels for failure designation. f_i = total number of failures of a given severity level in a given time interval, i=1, ...

A3.3 Implementation. Plot cumulative failures versus a suitable time base. The curve can be derived for the system as a whole, subsystems, or modules.

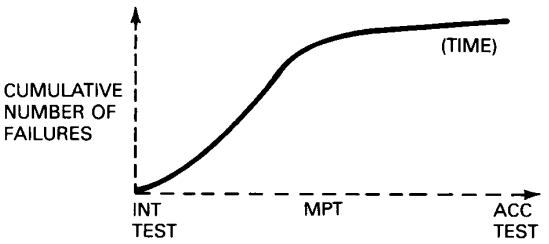
A3.4 Interpretation. The shape of the cumulative failure curve can be used to project when testing will be complete, provided that test coverage is sufficiently comprehensive. The curve gives an assessment of software reliability. It can provide an indication of clustering of faults in modules, suggesting a further concentration of testing for those modules. The total failures for each testing phase may be estimated by the asymptotic nature of the curve. A non-asymptotic curve indicates the need for continued testing.

A3.5 Considerations. Failures must be observed and recorded, and must be distinct. The failure curve may be compared with open SPR curves (Software Problem Report) or other measures. Several shapes for curves are proposed in the literature [A6, A9]. Profiles may be organized on a system, subsystem, or module basis. Test coverage must be both comprehensive and random.

A3.6 Training. Previous software testing experience is required if classification of severity and of failures is to be established. Good recordkeeping techniques are essential. Knowledge of the modular structure of the program and how it correlates with the test base is required in order to assess if the testing has been comprehensive.

A3.7 Example

Fig A3.7-1
Failure Profile



INT TEST = START OF INTEGRATION TEST.
MPT = MIDPOINT OF TESTING.
ACC TEST = START OF ACCEPTANCE TESTING.
SEE [A4, A6, A8, A9] FOR INTERPRETATIONS.

A3.8 Benefits. This measure is simple to use. Data can also be used as input to published reliability models.

A3.9 Experience. Many companies are using this measure and have developed their own guidelines but have not published their results [A8].

A3.10 References. See also Dobbins [A3, A4].

[A6] MENDIS, K. S. Quantifying Software Quality. *Quality Progress*, May 1982, pp 18-22.

[A7] MUSA, J. D., IANNINO, A., and OKUMOTO, K. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill, 1987.

[A8] SHOOMAN, M. L. *Software Engineering Design/Reliability/Management*. New York: McGraw Hill, 1983, pp 329-335.

[A9] TRACHTENBERG, M. Discovering How to Ensure Software Reliability. *RCA Engineer*, vol 27, no 1, Jan/Feb 1982, pp 53-57.

A4. Fault-Days Number

A4.1 Application. This measure represents the number of days that faults spend in the software system from their creation to their removal.

A4.2 Primitives

(1) Phase when the fault was introduced in the system.

(2) Date when the fault was introduced in the system.

(3) Phase, date, and time when the fault is removed.

FD_i = Fault days for the ith fault

NOTE: For more meaningful measures, time units can be made relative to test time or operational time.

A4.3 Implementation. For each fault detected and removed, during any phase, the number of days from its creation to its removal is determined (fault-days).

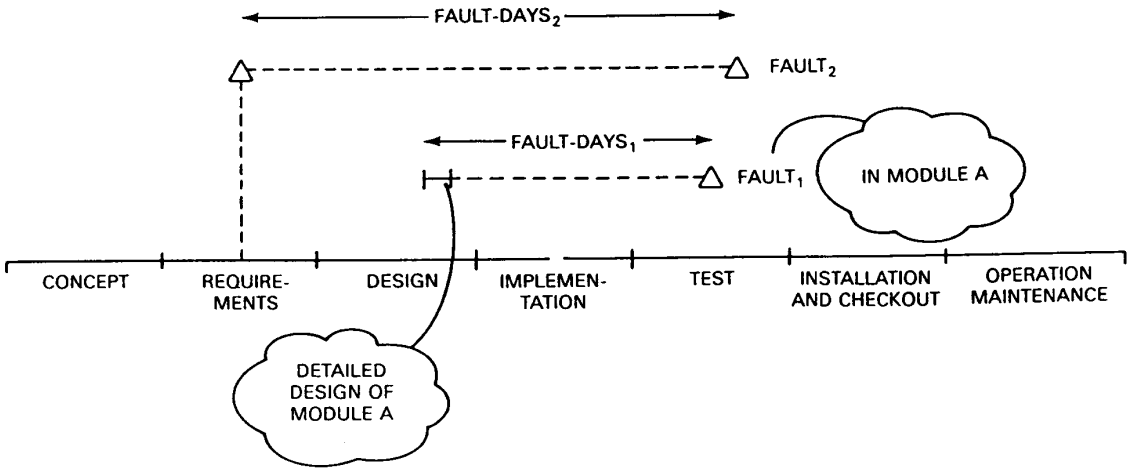
The fault-days are then summed for all faults detected and removed, to get the fault-days number at system level, including all faults detected/removed up to the delivery date. In cases when the creation date for the fault is not known, the fault is assumed to have been created at the middle of the phase in which it was introduced.

In Fig A4.3-1 the fault-days for the design fault for module A can be accurately calculated because the design approval date for the detailed design of module A is known. The fault introduced during the requirements phase is assumed to have been created at the middle of the requirement phase because the exact time that the corresponding piece of requirement was specified is not known.

The measure is calculated as follows:

$$\text{Fault days number } (FD) = \sum_i FD_i$$

Fig A4.3-1
Calculation of Fault-Days



A4.4 Interpretation. The effectiveness of the software design and development process depends upon the timely removal of faults across the entire life cycle. This measure is an indicator of the quality of the software system design and of the development process. For instance, a high number of fault days indicates either many faults (probably due to poor design, indicative of product quality), or long-lived faults, or both (probably due to poor development effort, indicative of process quality).

A4.5 Considerations. Careful collection of primitive data is essential to the successful use of this measure.

A4.6 Training. In order to improve the effectiveness of the software design and development process and to lower development costs (by timely detection and removal of faults), it is essential for personnel to become aware of the importance of recording data that would be needed to generate fault-birth information for each phase across the entire software life cycle phase (for example, date of module creation, changes, deletions, integration, etc, during the design phase).

A4.7 Example. Table 4.7-1 (extracted from Mills [A10], gives fault day information for two systems (A and B). These systems were developed with the same specifications and acceptance conditions, and had the same number of known faults at delivery time. However, the two systems possess very different histories in terms of faults. In this case, the knowledge of fault-days would have helped in predicting their very different behavior in terms of faults detected during acceptance.

These systems, built with the same specifications and subjected to the same acceptance testing, are not the same even though each has no known faults.

A4.8 Benefits. This measure encourages timely inspections and testing and can also assist management in improving the design and development process.

A4.9 Experience. Although limited published experience is available, this measure can be used in a software reliability program to monitor the quality of process and product development effort.

A4.10 Reference

[A10] MILLS, HARLAN D. Software Development. *IEEE Transactions on Software Engineering*, vol SE-4, no 4, Dec 1976.

A5. Functional or Modular Test Coverage

A5.1 Application. This measure is used to quantify a software test coverage index for a software delivery. The primitives counted may be either functions or modules. The operational user is most familiar with system functional requirements and will report system problems in terms of functional requirements rather than module test requirements. It is the task of the evaluator to obtain or develop the functional requirements and associated module cross reference table.

A5.2 Primitives

FE = number of the software functional (modular) requirements for which all test cases have been satisfactorily completed.

FT = total number of software functional (modular) requirements.

Table A4.7-1
Fault Days

During Development	A	Fault Days	B	Fault Days
Lines of code	50 000		50 000	
Faults fixed				
day old	100	100	500	500
week old	10	50	50	250
month old	5	100	50	1000
year old	5	1250	20	5000
Known faults	0		0	
Fault-days (total)		1500		6750
During acceptance				
Faults fixed	10		50	
Known faults	0		0	

A5.3 Implementation. The test coverage index is expressed as a ratio of the number of software functions (modules) tested to the total number of software functions (modules) that make up user (developer) requirements. This ratio is expressed as:

$$\text{Functional (modular) test coverage index} = \frac{FE}{FT}$$

A5.4 Interpretation. The test coverage index can be compared with the status of planned testing and the impact that functional (modular) changes are having on schedule and resources.

A5.5 Considerations. Additional data required to support the test coverage index include:

(1) Software version identification and date of release.
(2) The functions (modules) delivered, changed, and added for each release.
(3) Test results treated as additional primitives.

A5.6 Training. Training is important in methods that identify functions and modules. Examples of such methods are given in Ceriani, Cicu, and Maiocchi [A11].

A5.7 Example. Table A5.7-1 depicts an example of the proposed test data and resulting test coverage index for a series of software releases.

A5.8 Benefit. The test coverage index provides a measure of the percentage of software tested at any time. For many applications programs, the function (modular) test coverage index might always be relatively low because of the system application (eg, automatic test equipment) and the restrictions of the test environment (eg, electronic warfare systems).

A5.9 Experience. Application references for the test coverage index are contained in Henderson [A12] and Maiocchi, Mazzetti, and Villa [A13].

A5.10 References

[A11] CERIANI, M., CICU, A., and MAIOCCHI, M. A Methodology for Accurate Software Test Specification and Auditing. *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, eds, North Holland Publishing Company, 1981.

[A12] HENDERSON, JAMES B. Hierarchical Decomposition: Its Use in Testing. *Testing Techniques Newsletter*, vol 3, no 4, San Francisco: Software Research Associate, Nov 1980.

[A13] MAIOCCHI, M., MAZZETTI, A., and VILLA, M. TEFAX: An Automated Test Factory for Functional Quality Control of Software Projects. Colloque de Genie Logiciel #2, Nice, 1984.

[A14] MILLER, E. Coverage Measure Definition Reviewed. *Testing Techniques Newsletter*, vol 3, no 4, San Francisco: Software Research Associate, Nov 1980.

Table A5.7-1
Example of Software Test Coverage Index Evaluation

Release: Date:	1 3/82	2 5/82	3 8/82	4 12/82	5 2/83	6 5/83	7 Prod
Identification of deficiency corrected	n/a	n/a	#3	#15 #16	#17	#20	#25
Modules:	99	181	181	181	183	184	185
Changes:	13	6	2	36	23	Est(6)	Est(4)
Additions:	82	0	0	2	1	Est(1)	Est(0)
Deletions:	0	0	0	0	0	Est(0)	Est(0)
Module testing completed: (FE)	0	50	100	100	110	TBD	TBD
Module requirements: (FT)	181	181	181	183	184	Est(185)	Est(185)
Test coverage index	0	.28	.55	.55	.60	TBD	TBD

NOTE: This table uses modules as the counted quantities instead of user functions. A similar table would be generated for user functions as the quantities counted.

A6. Cause and Effect Graphing

A6.1 Application. Cause and effect graphing aids in identifying requirements that are incomplete and ambiguous. This measure explores the inputs and expected outputs of a program and identifies the ambiguities. Once these ambiguities are eliminated, the specifications are considered complete and consistent.

Cause and effect graphing can also be applied to generate test cases in any type of computing application where the specification is clearly stated (ie, no ambiguities) and combinations of input conditions can be identified. It is used in developing and designing test cases that have a high probability of detecting faults that exist in programs. It is not concerned with the internal structure or behavior of the program.

A6.2 Primitives

List of causes: distinct input conditions.

List of effects: distinct output conditions or system transformation (effects are caused by changes in the state of the system)

$A_{existing}$ = number of ambiguities in a program remaining to be eliminated

A_{tot} = total number of ambiguities identified

A6.3 Implementation. A cause and effect graph is a formal translation of a natural language specification into its input conditions and expected outputs. The graph depicts a combinatorial logic network.

To begin, identify all requirements of the system and divide them into separate identifiable entities. Carefully analyze the requirements to identify all the causes and effects in the specification. After the analysis is completed, assign each cause and effect a unique identifier. For example, $E1$ for effect one or $I1$ for input one.

To create the cause and effect graph, perform the following steps:

(1) Represent each cause and each effect by a node identified by its unique number.

(2) Interconnect the cause and effect nodes by analyzing the semantic content of the specification and transforming it into a Boolean graph. Each cause and effect can be in one of two states: true or false. Using Boolean logic, set the possible states of the causes and determine under what conditions each effect will be present.

(3) Annotate the graph with constraints describing combinations of causes and effects that are impossible because of semantic or environmental constraints.

(4) Identify as an ambiguity any cause that does not result in a corresponding effect, any effect that does not originate with a cause as a source, and any combination of causes and effects that are inconsistent with the requirement specification or impossible to achieve.

The measure is computed as follows:

$$CE(\%) = 100 \times \left(1 - \frac{A_{existing}}{A_{tot}}\right)$$

To derive test cases for the program, convert the graph into a limited entry decision table with "effects" as columns and "causes" as rows. For each effect, trace back through the graph to find all combinations of causes that will set the effect to be TRUE. Each such combination is represented as a column in the decision table. The state of all other effects should also be determined for each such combination. Each column in the table represents a test case.

A6.4 Interpretation. When all of the causes and effects are represented in the graph and no ambiguities exist, the measure is 100%. A measure of less than 100% indicates some ambiguities still exist that have not been eliminated. Some of these ambiguities may result from incomplete graphing of cause and effect relationships or from incomplete statements in the natural language specification. Such ambiguities should be examined and reconciled.

The columns in the limited entry decision table should be used to create a set of test cases. For each test case:

(1) Establish input values or conditions for each cause that sets the effect to be true;

(2) Determine the expected value or result that should be demonstrated when the effect is true;

(3) Consider boundary conditions, incorrect values, etc.

A6.5 Considerations. Cause and effect graphing is a technique used to identify incompleteness and ambiguities in program specifications. It works best with structured specifications. It has the added capability of producing a useful set of test cases. However, this technique does not produce all the useful test cases that might be identified. It also does not adequately explore

boundary conditions. This technique should be supplemented by other test design techniques addressing these areas. The graph can be processed mechanically to provide test cases.

Manual application of this technique is a tedious, long, and moderately complex process. However, the technique could be applied to selected modules with complex conditional logic. The requirements specification must be complete.

A6.6 Training. The requirement specification of the system must be clearly understood in order to successfully apply the graphing techniques to eliminate ambiguities.

Cause and effect graphing is a mathematically based technique that requires some knowledge of Boolean logic.

A6.7 Example. A database management system requires that each file in the database has its name listed in a master index that identifies the location of each file. The index is divided into ten sections. A small system is being developed that will allow the user to interactively enter a command to display any section of the index at a terminal. Cause and effect graphing is used to identify ambiguities in the specifications and to develop a set of test cases for the system.

The requirements specification for this system is as follows: To display one of ten possible index sections, a command must be entered consisting of a letter and a digit. The first character entered must be a "D" (for display) or an "L" (for list) and it must be in column 1. The second character entered must be a digit (0-9) in column 2. If this command occurs, the index section identified by the digit is displayed on the terminal. If the first character is incorrect, error message A is printed. If the second character is incorrect, error message B is printed.

The error messages are:

A = INVALID COMMAND

B = INVALID INDEX NUMBER

The causes and effects are identified as follows (see Fig A6.7-1):

Causes

(1) Character in column 1 is a "D"

(2) Character in column 1 is an "L"

(3) Character in column 2 is a digit.

Effects

(50) Index section is displayed

(51) Error message A is displayed

(52) Error message B is displayed.

Figure A6.7-1 was constructed through analysis of the semantic content of the specification.

In Fig A6.7-1, node 20 is an intermediate node representing the Boolean state of node 1 or node 2. The state of node 50 is true if the state of nodes 20 and 3 are both true. The state of node 20 is true if the state of node 1 or node 2 is true. The state of node 51 is true if the state of node 20 is not true. The state of node 52 is true if the state of node 3 is not true. Nodes 1 and 2 are also annotated with a constraint that nodes 1 and 2 cannot be true simultaneously (the exclusive constraint).

An ambiguity in the requirements specification results from the statement that the index section is displayed on the terminal. The specification is not clear whether the index section is listed on the printer or displayed when a valid command (eg, L7) occurs. To eliminate this ambiguity, redefine effect 50 to be effect 50A (index section is displayed), and effect 50B (index section is listed).

To derive the test cases, the graph is converted into a decision table as shown in Table A6.7-1. For each test case, the bottom of the table indicates

Fig A6.7-1
Boolean Graph

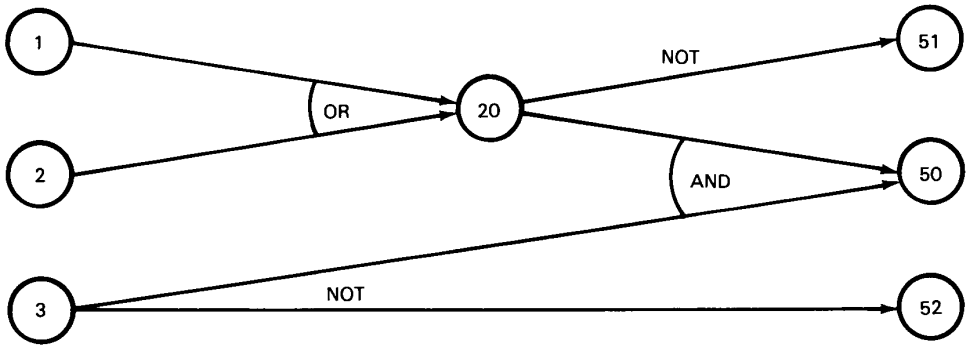


Table A6.7-1
Decision Table

Cause	Effect						
	20			50A	50B	51	52
1	1	1	0	1			
2	1	0	1		1		
20						0	
3				1	1		0
*Not allowed	*	1	1	1	1	1	1

Table A6.7-2
Test Cases

Test Case #	Inputs	Expected Results
1	*D5*	Index section 5 is displayed
2	*L4*	Index section 4 is listed
3	*B2*	Invalid command
4	*DA*	Invalid index number

which effect will be present (indicated by a 1). For each effect, all combinations of causes that will result in the presence of the effect is represented by the entries in the columns of the table. Blanks in the table mean that the state of the cause is irrelevant.

Last, each column in the decision table is converted into test cases. Table A6.7-2 gives one example of such an approach.

A6.8 Benefits. This measure provides thorough requirements analysis, elimination of ambiguities, and a good basis for test case development.

A6.9 Experience. This technique has a modest amount of experience.

A6.10 References

[A15] ELMENDORF, W. R. *Cause-Effect Graphs on Functional Testing.* Poughkeepsie: IBM Systems Development Division, TR-00.2487, 1973.

[A16] MYERS, GLENFORD J. *The Art of Software Testing.* New York: Wiley-Interscience, 1979.

[A17] POWELL, B. P., ed. *Validation, Verification, and Testing Technique and Tool Reference Guide.* National Bureau of Standards Special Publication 500-93, 1982.

A7. Requirements Traceability

A7.1 Application. This measure aids in identifying requirements that are either missing from, or in addition to, the original requirements.

A7.2 Primitives

$R1$ = number of requirements met by the architecture.

$R2$ = number of original requirements.

A7.3 Implementation. A set of mappings from the requirements in the software architecture to the original requirements is created. Count each requirement met by the architecture ($R1$) and count each of the original requirements ($R2$). Compute the traceability measure:

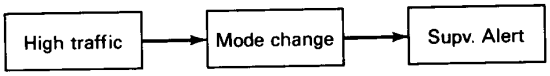
$$TM = \frac{R1}{R2} \times 100\%$$

A7.4 Interpretation. When all of the original software requirements are covered in the software architecture the traceability measure is 100%. A traceability measure of less than 100% indicates that some requirements are not included in the software architecture. The architecture may contain requirements beyond the original requirements. Some of these may result naturally from stepwise refinement. Others may result from new requirements being added to the architecture. Such cases should be examined and reconciled.

A7.5 Considerations. Stepwise refinement of the requirements into the architecture produces a natural set of mappings from which to derive the measure. For large systems, automation is desirable.

A7.6 Training. Training is needed to properly interpret the results, and to make suitable decisions about reconciling architectural issues.

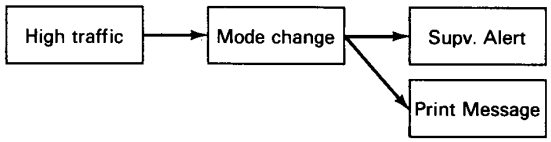
A7.7 Example. Suppose there is a requirement to send an alert to the supervisor console, and to print a hard copy of the alert whenever high traffic mode is entered. The architecture for such a requirement might be as follows:



In this architecture, the hardcopy is not shown, so that complete traceability is not obtained, and

the traceability will be less than 100%, with $R2 = R1 + 1$.

Alternatively, assume that there is a requirement to send an alert to the supervisor console whenever high traffic mode is entered. The architecture for this requirement might be as follows:



In this architecture, an additional requirement has been introduced that was not in the original requirements. This would result in a traceability error also, with traceability greater than 100%, and $R1 = R2 + 1$.

A7.8 Benefits. The requirements traceability measure applied to the software architecture phase can aid in identifying requirements that have not been accounted for in the architecture.

A7.9 Experience. This technique has been used extensively on large software projects [A18].

A7.10 References

[A18] HENNINGER, K. Specifying Software Requirements for Complex Systems, New Techniques and Their Application. *IEEE Transactions on Software Engineering*, vol SE-6, no 1, Jan 1980, pp 1-14.

[A19] PERRIENS, M. P. *Software Requirements Definition and Analysis with PSL and QBE.* IBM Technical Report FSD 80-0003, Bethesda: IBM Federal Systems Division, Oct 1980.

[A20] YEH, R., ZAVE, P., CONN, A., and COLE, G., Jr. *Software Requirements: A Report on the State of the Art.* Computer Science Technical Report Series TR-949, College Park, Maryland: University of Maryland, Oct 1980.

A8. Defect Indices

A8.1 Application. This measure provides a continuing, relative index of how correct the software is as it proceeds through the development cycle. This measure is a straightforward, phase dependent, weighted, calculation that requires no knowledge of advanced mathematics or statistics. This measure may be applied as early in the life cycle as the user has products that can be evaluated.

A8.2 Primitives. The following list applies to each phase in the life cycle:

- D_i = Total number of defects detected during the ith phase.
- S_i = Number of serious defects found.
- M_i = Number of medium defects found.
- PS = Size of the product at the ith phase.
- T_i = Number of trivial defects found.
- W_1 = Weighting factor for serious defects (default is 10).
- W_2 = Weighting factor for medium defects (default is 3).
- W_3 = Weighting factor for trivial defects (default is 1).

A8.3 Implementation. The measure is generated as a sum of calculations taken throughout development. It is a continuing measure applied to the software as it proceeds from design through final tests.

At each phase of development, calculate the phase index (PI_i) associated with the number and severity of defects.

$$PI_i = W_1 \frac{S_i}{D_i} + W_2 \frac{M_i}{D_i} + W_3 \frac{T_i}{D_i}$$

The defect index (DI) is calculated at each phase by cumulatively adding the calculation for PI_i as the software proceeds through development:

$$DI = \Sigma (i * PI_i)/PS$$
$$= (PI_1 + 2PI_2 + \ldots + iPI_i + \ldots)/PS$$

where each phase is weighted such that the further into development the software has progressed, such as phase 2 or 3, the larger the weight (ie, 2 or 3, respectively) assigned.

The data collected in prior projects can be used as a baseline figure for comparison.

A8.4 Interpretation. The interpretation is based on the magnitude of the defect index: the smaller the number, the better. Each organization or project should compute baseline numbers from past projects for each of the types of software produced. Then, each project should be tracked to assess the following:

(1) Whether performance has increased.
(2) Impact of programmer education programs.
(3) Impact of new programming practices.
(4) Impact of entering a new business area or performing new, unfamiliar tasks.
(5) Impact of mid-stream changes, such as major requirement changes.

This measure can be used as adjunct to either fault assessment or error cause analysis. Each user may determine a unique defect classification. For example, serious defects may be described as faults that keep the software from functioning. Medium defects may be described as faults that will cause the software to operate in a degraded mode. Trivial defects may be defined as those that will not affect functional performance, such as misspelled words on displays.

A8.5 Considerations. This measure is best applied when used with other measures because it gives an indication of what's happening, but not why. Error cause analysis in conjunction with defect index calculations can be quite enlightening.

A8.6 Training. Little or no measurement training is required.

A8.7 Example. For corresponding phases of development, calculated for three different projects, the results may be

$DI_1 = 15 \qquad DI_2 = 20 \qquad DI_3 = 205$

For these projects, the results show that project three has some significant problems as compared to measures of projects one and two taken at corresponding phases of development. It indicates that management action is clearly required in order to ascertain the difficulty.

A8.8 Benefits. This measure is both a process and a product measure.

As a process measure it provides an indication of whether the defect removal process employed at a given phase of development was employed in a sufficiently effective manner. It also affords the developer a means of assessing the effectiveness of phase related defect removal processes.

As a product measure, it allows the development management a means of assessing the degree of correctness of the software at a given phase, thereby providing an indication of the reliability growth as the product moves through the development phases toward final acceptance.

It is easy to use and interpret and is successively applicable across development phases. It can be done manually or automated. It can also be easily expanded to include sub-phases, with each sub-phase separately weighted. The index is weighted so that the more defects found early, the better the result. Use of the index thereafter encourages the application of early defect removal processes such as inspections.

If requirements are done in a structural requirements language, inspections can be started at the requirements level and this defect index (*DI*) measure can be adjusted to begin at the requirements phase.

A8.9 Experience. The experience base for this measure is small.

A8.10 Reference

[A21] SCHULMEYER, G., ed. Handbook for Software Quality Assurance. New York: Van Nostrand Publishing Company, 1985.

A.9 Error Distribution(s)

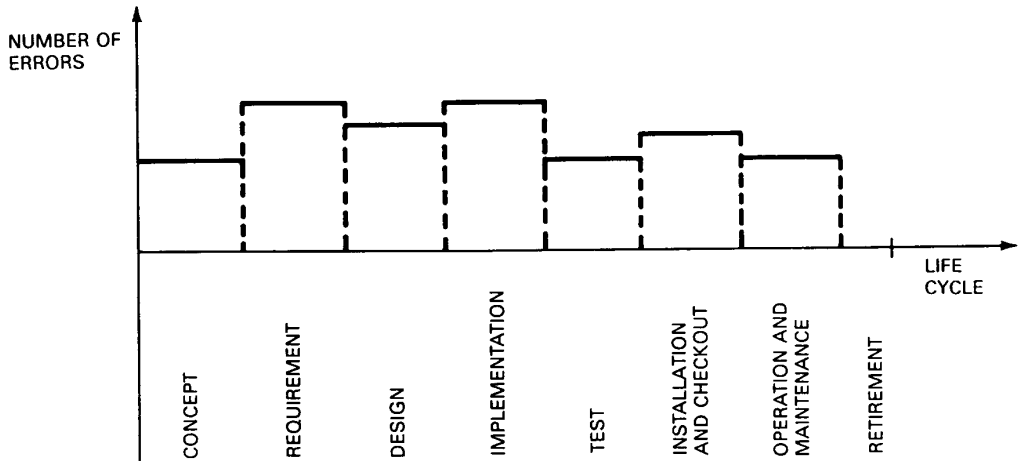
A9.1 Application. The search for the causes of software faults and failures involves the error analysis of the defect data collected during each phase of the software development. Distribution of the errors allows ranking of the predominant failure modes.

A9.2 Primitives. Error description notes the following points:

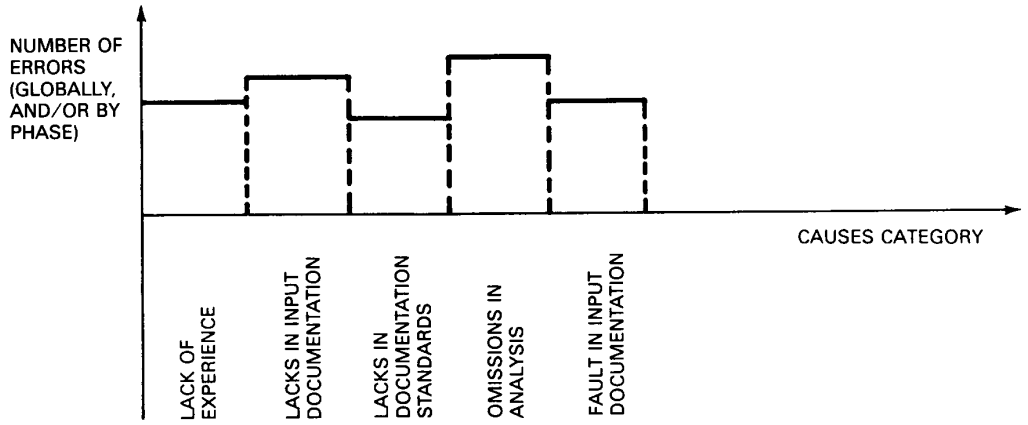
(1) Associated faults
(2) Types
(3) Severity
(4) Phase introduced
(5) Preventive measure
(6) Discovery mechanism including reasons for earlier non-detection of associated faults

A9.3 Implementation. The primitives for each error are recorded and the errors are counted according to the criteria adopted for each classification. The number of errors are then plotted for each class. Examples of such distribution plots are shown in Fig A9.3-1. In the three examples of Fig A9.3-1, the errors are classified and counted by phase, by the cause and by the cause for deferred fault detection. Other similar classifications could be used, such as the type of steps suggested to prevent the reoccurrence of similar errors or the type of steps suggested for earlier detection of the corresponding faults.

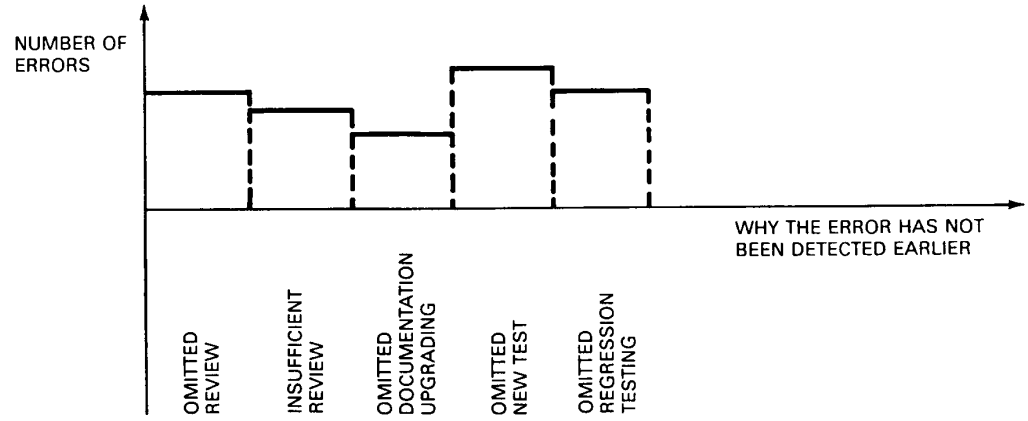
A9.4 Interpretation. Plots similar to Fig A9.3-1 could be analyzed by management to identify the phases that are most error prone, and to identify causes, risks, and recommend the nature and priority of corrective actions to improve error avoidance.



(a) Error Distribution by Phase



(b) Error by Cause Category



(c) Suggested Causes for Error Detection Deferral

Fig A9.3-1
Error Analysis

This measure is also useful to recommend steps to prevent the recurrence of similar errors, to eliminate known causes, and to detect faults earlier for purpose of reducing software life cycle costs.

A9.5 Considerations. Error analysis requires effective support and participation by management, because the errors usually have their origin in deficient procedures, methods, and techniques for design, inspection, and testing.

A9.6 Training. A global knowledge of the effective methodologies for the software development process for the entire life cycle phase is required to support effective error analysis and interpretation.

A9.7 Example. See Fig A9.3-1.

A9.8 Benefits. Error analysis by distributions is a key factor in understanding the areas where interventions or corrective actions or both are the most effective steps towards error and fault avoidance.

A9.9 Experience. Although there is little experience in the literature related to this measure, it is recommended as an important method for analyzing causes of errors and error prone areas.

A9.10 References

[A22] ANDRES, A. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, Jun 1975 (reprinted in *Tutorial on Models and Metrics for Software Management and Engineering*). IEEE catalog no EHO-16-7, IEEE Computer Society, Sept 1980.

[A23] RZEVSKI, G. Identification of Factors which Cause Software Failures. Proceedings of Annual Reliability and Maintainability Symposium, Los Angeles, Jan 26-28, 1982.

A10. Software Maturity Index

A10.1 Application. This measure is used to quantify the readiness of a software product. Changes from a previous baseline to the current baseline are an indication of the current product stability. A baseline can be either an internal release or an external delivery.

A10.2 Primitives

M_T = Number of software functions (modules) in the current delivery

F_c = Number of software functions (modules) in the current delivery that include internal changes from a previous delivery

F_a = Number of software functions (modules) in the current delivery that are additions to the previous delivery

F_{del} = Number of software functions (modules) in the previous delivery that are deleted in the current delivery

A10.3 Implementation. The software maturity index (*SMI*) may be calculated in two different ways, depending on the available data (primitives).

A10.3.1 Implementation 1

(1) For the present (just delivered or modified) software baseline, count the number of functions (modules) that have been changed (F_c).

(2) For the present software baseline, count the number of functions (modules) that have been added (F_a) or deleted (F_{del}).

(3) For the present software baseline, count the number of functions (modules) that make up that release (M_T).

Calculate the maturity index using the following equation:

$$\text{Maturity Index} = \frac{\text{Number of current functions (modules)} - \left(\text{Number of current baseline functions (modules) that have been added} + \text{Number of functions (modules) changed} + \text{Number of current baseline functions (modules) that have been deleted}\right)}{\text{Number of current functions (modules)}}$$

ie, $$SMI = \frac{M_T - (F_a + F_c + F_{del})}{M_T}$$

Notice that the total number of current functions (modules) equals the number of previous functions (modules) plus the number of current baseline functions (modules) that were added to the previous baseline minus the number of functions (modules) that were deleted from the previous baseline. In the software maturity index calculation, the deletion of a function (module) is treated the same as an addition of a function (module).

A10.3.2 Implementation 2. The software maturity index may be estimated as

$$SMI = \frac{M_T - F_c}{M_T}$$

The change and addition of functions (modules) is tracked and the maturity index is calculated

for each baseline. Problem reports that would result in the software update subsequent to the tracking period are included in the maturity analysis by estimating the configuration of the subsequent releases.

A10.4 Interpretation. The maturity index for each software release can be tracked to determine the relative impact that changes and additions are having on the software configuration and whether the software is approaching a stable configuration.

A10.5 Considerations. Additional data that is required in conjunction with the maturity index measures are as follows:

(1) The impact software function (module) changes are having on previously completed testing (see A5, Functional or Modular Test Coverage).
(2) The software release identification and date of release.
(3) Identification of the customer (producer) problems that are corrected in the release.
(4) Estimate of the configuration changes that would be required to correct the remaining customer (producer) identified problems.

The maturity index is used as a primitive in software release readiness (see A34). The functional test coverage must also be calculated in conjunction with the maturity index (see A5).

A10.6 Training. The software training required would include identification of the user requirements and the associated software functions and modules that implement the requirements. The operational user must be familiar with the system functional requirements in order to report problems in terms of those requirements.

A10.7 Example. Table A10.7-1 depicts the data and resulting maturity index for a series of software releases. Keep in mind that a series of maturity index values determined throughout a test program will be misleading unless the corresponding functional test coverage index for each release is also determined.

A10.8 Benefits. The maturity index provides a measure of the relative impact that software changes and additions are making to the total software configuration. In addition, the maturity index provides an indication of the stability of the software throughout the period of user test and evaluation.

A10.9 Experience. See Ceriani, Cicu, and Maiocchi [A11] for related experience.

A11. Manhours per Major Defect Detected

A11.1 Application. The design and code inspection processes are two of the most effective defect removal processes available. The early removal of defects at the design and implementation phases, if done effectively and efficiently, significantly improves the reliability of the developed product and allows a more controlled test environment. This measure provides a quantitative figure that can be used to evaluate the efficiency of the design and code inspection processes.

A11.2 Primitives

T_1 = Time expended by the inspection team in preparation for design or code inspection meeting.

Table A10.7-1
Example Results of a Software Maturity Index Evaluation

Release: Date:	1 3/82	2 5/82	3 8/82	4 12/82	5 2/83	6 5/83	7 Prod
Identification of deficiency corrected	n/a	n/a	#3	#15 #16	#17	#20	#25
Modules:	99	181	181	181	183	184	185
Changes:	13	6	2	36	23	Est(6)	Est(4)
Additions	82	0	0	2	1	Est(1)	Est(0)
Deletions	0	0	0	0	0	Est(0)	Est(0)
Maturity index	.475	.966	.99	.79	.868	(.96)	(.98)
	181-95	181-6	179	181-38	183-24	184-6	181
	181	181	181	181	183	184	185

T_2 = Time expended by the inspection team in conduct of a design or code inspection meeting.
S_i = Number of major (non-trivial) defects detected during the ith inspection.
I = Total number of inspections to date.

A11.3 Implementation. At each inspection meeting, record the total preparation time expended by the inspection team. Also, record the total time expended in conducting the inspection meeting. All defects are recorded and grouped into major/minor categories. (A major defect is one that must be corrected for the product to function within specified requirements.)

The inspection times are summarized and the defects are cumulatively added. The computation should be performed during design and code. If the design is not written in a structural design language, then this measure can be only applied during the implementation phase.

The manhours per major defect detected is

$$M = \frac{\sum_{i=1}^{I} (T_1 + T_2)_i}{\sum_{i=1}^{I} S_i}$$

This computation should be initiated after approximately 8,000 lines of detailed design or code have been inspected.

A11.4 Interpretation. This measure is applied to new code development, not to modification of an old code, unless the modification is a large block change.

From empirical experience, the measure (M) for new code shall fall consistently between three and five. If significant deviation from this range is experienced, the matter should be investigated. If the measure is very low, the defect density may be too high and a major product examination may be in order. If the measure is very high, the inspection process is very likely not being properly implemented and, therefore, the anticipated impact on reliability improvement expected from the inspection will not be realized.

A11.5 Considerations. This measure assumes a knowledge of the design and code inspection process. It further assumes consistent, standard data collection and recording. Experience has shown that the inspection process is quite effective, if it is strongly committed to and supported by management. The earlier the inspection process can be applied, the greater the benefit derived. It is therefore recommended that program design languages be used.

A11.6 Training. Little or no training is required for calculation of the measure. Inspections require trained personnel.

A11.7 Example

$$\left.\begin{array}{l} I = 2 \\ T_1 = 28 \\ T_2 = 25 \\ S_1 = 16 \end{array}\right\} i=1 \qquad \left.\begin{array}{l} T_1 = 40 \\ T_2 = 50 \\ S_2 = 4 \end{array}\right\} i=2$$

$$M = \frac{28 + 25 + 40 + 50}{16 + 4} = \frac{143}{20} = 7.15$$

A11.8 Benefits. This measure is easy to compute. It serves as one of the index parameters to evaluate the effectiveness and efficiency of the inspection process. Deviations from the norm serve as an indicator that immediate attention is required and, therefore, gives the developer a change to make whatever modifications are necessary to maintain sufficient product correctness and reliability during early development.

A11.9 Experience. Utilization of the inspection process has been found to be highly effective in defect removal and correction.

A11.10 References. See also Dobbins and Buck [A3, A4].

[A24] FAGAN, MICHAEL E. Design and Code Inspection to Reduce Errors in Program Development. *IBM Systems Journal*, vol 15, no 3, Jul 1976, pp 182-211.

A12. Number of Conflicting Requirements

A12.1 Application. This measure is used to determine the reliability of a software system, resulting from the software architecture under consideration, as represented by a specification based on the entity-relationship-attribute model [A19].

A12.2 Primitives

(1) List of the inputs
(2) List of the outputs
(3) List of the functions performed by the program

A12.3 Implementation. The mappings from the software architecture to the requirements are identified. Mappings from the same specification item to more than one differing requirement are examined for requirements inconsistency. Mappings from more than one specification item to a single requirement are examined for specification inconsistency.

A12.4 Interpretation. If the same specification item maps to two different requirements items, the requirements should be identical. Otherwise, the requirements are inconsistent. If more than one specification item maps to a single requirement, (this should result only from application of refinement techniques) the specification items should be checked for possible inconsistency.

A12.5 Considerations. Stepwise refinement or structured analysis of the requirements into the architecture produces the best results, since there is likely to be a natural set of mappings resulting from this process. For large systems, automation is desirable.

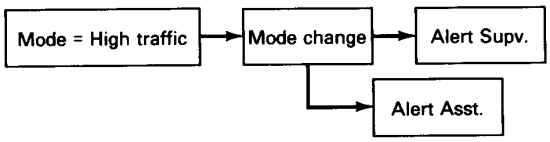
A12.6 Training. Training is needed to interpret the results, and make suitable decisions on the adequacy of the architecture and the requirements. Experience/training in requirements analysis, the entity-relationship-attribute model, and software specification techniques are needed [A20].

A12.7 Example. Consider an example taken from the air traffic control field. Suppose there are two distinct requirements for action to be taken in the event of high traffic. These requirements might be stated as:

(1) In the event of high traffic, send an operator alert to the supervisor console. If the supervisor console is busy, then send to the assistant.

(2) In the event of high traffic, send an operator alert to the assistant supervisor console.

In the architecture, this could be implemented as:



A PDL representation might be as follows:

```
If mode = high traffic
  Send operator alert to supervisor console
  Send operator alert to assistant console
Endif
```

This example represents a potential conflicting requirement between sending an alert to both the supervisor and the assistant. This would need to be examined and resolved.

A12.8 Benefits. Identification of conflicting requirements during the software architecture phase can avoid problems later in the life cycle.

A12.9 Experience. Many large software projects informally use these techniques for determining reliability [A18].

A13. Number of Entries and Exits per Module

A13.1 Application. This measure is used to determine the difficulty of a software architecture, as represented by entry/exit points identified in a modular specification/design language.

A13.2 Primitives

e_i = number of entry points for the ith module
x_i = number of exit points for the ith module

A13.3 Implementation. The number of entry points and exit points for each module is

$$m_i = e_i + x_i$$

A13.4 Interpretation. Each module should have one entry and one exit point for each major function. Additional exit points may be allowed for error handling. The number of functions per module should be limited. A suggested maximum number of functions per module is five.

A13.5 Considerations. Data encapsulation should be used to optimize the number of functions and entry/exit points. If data is properly encapsulated [A27], the number of entry and exit points for each module function will be small. For large systems, automation of the count of entries and exits is desirable, so the user does not need to manually examine each module.

A13.6 Training. In both the manual and automated situations, training is needed to interpret the results and make suitable trade-off decisions.

A13.7 Example. Given a module specification (Fig A13.7-1), this method can be used. This module has four entry points, and, since it is structured, four exit points.

```
mod idset
    def
        hash (id) = return an index value from 1 to max as a f(id)
        type item = rec
                        id: idtype
                        data: datatype
                    cer
ism
    input
        var in: item
        var searchid: idtype
    output
        var out:  item
        var status: (OK, full, notthere, alreadythere)
    state
        const max = [maximum size of set]
        var s: set of item [s ≤ max]
    init
        [s:=φ]
    trans
        entry add (in in, out status)
            [ (i∈s)in.id=i.id → status: = alreadythere
              true → ( s <max → s, status: = s∪ (in),OK
                          true → status: = full)]
        entry delete (in Searchid, out status)
            [ (i∈s) searchid = i.id → s,status: = s-(in),OK
              true → status: = notthere]
        entry has (in searchid, out status, out)
            [ (i∈s) searchid = i.id → out, status: = i,OK
            true → status: = notthere]
        entry clear
            [s:=φ]
des
    (*to be added*)
dom
```

Fig A13.7-1
Program Design Language Module Specification

A13.8 Benefits. The entry/exit point measure applied to the software architecture phase can aid in identifying modules that do not exhibit proper data encapsulation, and are likely to result in high operating system overhead times (which occur when one module invokes another) and difficulty in testing.

A13.9 Experience. Minimizing the number of entry/exit points is a key feature of structured design and structured programming techniques that have been used in software architecture definition. Limited published experience is currently available.

A13.10 References

[A25] GUTTAG, J. Abstract Data Types and the Development of Data Structures. Conference on Data, Salt Lake City, 1976. *CACM*, vol 20, no 6, June 1977, pp 396-404.

[A26] LISKOV, B. H. and ZILLES, S. N. Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering*, vol SE-2, no 1, Mar 1975, pp 7-19.

[A27] PARNAS, D. L. On the Criteria To Be Used in Decomposing Systems into Modules. *CACM*, vol 15, no 12, 1972, pp 1053-1058.

A14. Software Science Measures

A14.1 Application. These measures apply Halstead software science to the properties and structure of computer programs. They provide measures of the complexity of existing software, predict the length of a program, and estimate the amount of time an average programmer can be expected to use to implement a given algorithm.

This measure computes the program length by counting operators and operands. The measure suggests that the difficulty of a given program can be derived, based on the above counts.

A14.2 Primitives

n_1 = Number of distinct operators in a program

n_2 = Number of distinct operands in a program

N_1 = Total number of occurrences of the operators in a program

N_2 = Total number of occurrences of the operands in a program

A14.3 Implementation. Once the source code has been written, this measure can be applied to predict program difficulty and other derived quantities using the following equations (Halstead [A30]).

Program vocabulary: $l = n1 + n2$

Observed program length: $L = N1 + N2$

Estimated program length: $\hat{L} = n1\ (\log_2 n1) + n2\ (\log_2 n2)$

Program volume: $V = L\ (\log_2 l)$

Program difficulty: $D = (n1/2)\ (N2/n2)$

Program level: $L1 = 1/D$

Effort: $E = \dfrac{V}{L1}$

Number of errors: $B = \dfrac{V}{3000} \simeq \dfrac{E^{2/3}}{3000}$

Time: $T = E/S$ (S = Stroud number; typical value is eighteen elementary mental discriminations per second)

An alternate expression to estimate program length is the factorial length estimator (A32):

$\hat{L} = \log_2((n1)!) + \log_2((n2)!)$

A14.4 Interpretation. The interpretation of these measures is illustrated in the example (A14.7); for the average PL/I program, difficulty should be less than 115. Difficulty of greater than 160 indicates serious problems. Difficulty data is available for other languages as well [A29.5, A30, A35].

A14.5 Considerations. The technique is recommended only if automated support is available. The software needed to parse the programs and accumulate data is somewhat lengthy. However, there has been some experience in writing such software.

A14.6 Training. Training is needed in the definition of operators and operands and in the interpretation of the results. The results will differ depending on the language and applications. The user should examine results obtained in some of the reference material beforehand, and, if possible, contact other users.

A14.7 Example

Program fragment (written in *PL/I*):

```
/* x becomes the maximum of A and B*/
  if A > B
  then X = A;
  else X = B;
```

Software science basic parameters:

operators and their frequencies: $n1 = 4$ $N1 = 6$

if...then...else	1
>	1
=	2
;	2

operands and their frequencies: $n2 = 3$ $N2 = 6$

A	2
B	2
X	2

Software science measurements:

Program vocabulary: $l = n1 + n2 = 4 + 3 = 7$

Observed program length: $L = N1 + N2 = 6 + 6 = 12$

Estimated program length:

$$\hat{L} = n1 (\log_2 n1) + n2 (\log_2 n2)$$
$$= 4(\log_2 4) + 3 (\log_2 3) = 12.76$$

Estimated program length using the factorial length estimator is

$$\hat{L} = \log_2((4)!) + \log_2((3)!) = 7.17$$

Program volume: $V = L (\log_2 l) = 12 (\log_2 7) = 33.68$

Program difficulty: $D = (n1/2) (N2/n2) = (4/2) (6/3) = 4.0$

Program level: $L1 = \frac{1}{D} = \frac{1}{4.0} = .25$

Effort: $E = V/L1 = 33.68/.25 = 134.72$

Number of errors: $B = V/3000 = 33.68/3000 = 0.01123$

Time: $T = E/S = 134.72/18 = 7.48$

A14.8 Benefit. Application of the software science measures to the implementation phase can aid in the identification of programs that are overly complex, and likely to result in excessive development effort.

A14.9 Experience. There has been considerable and notable experience reported concerning this measure.

A14.10 References

[A28] CONTE, S. D., DUNSMORE, H. E., and SHEN, V. Y. *Software Engineering Metrics and Models.* Menlo Park: Benjamin/Cummings Publishing Co, 1986.

[A29] COULTER, M. S. Software Science and Cognitive Psychology. *IEEE Transactions on Software Engineering*, vol SE-9, no 2, Mar 1983, pp 166-171.

[A29.5] FITZSIMMONS, A., and LOVE, T. A Review and Evaluation of Software Science. *ACM Computing Surveys*, vol 10, no 1, Mar 1978, pp 3-18.

[A30] HALSTEAD, M. H. *Elements of Software Science.* Operating and Programming Systems Series, P. J. Denning, ed, New York: Elsevier North Holland, 1977.

[A31] HALSTEAD, M. H., GORDON, R. D., and ELSHOFF, J. L. *On Software Physics and GM's PL/I Programs.* GM Research Publication GMR-2175, Warren, Michigan: GM Research Laboratories, Jun 1976.

[A32] JENSEN, H. J., and VAIRAVAN, K. An Experimental Study of Software Metrics for Real Time Software. *IEEE Transactions on Software Engineering*, vol SE-11, no 2, Feb 1985, pp 231-234.

[A33] LIND, R. K., and VAIRAVAN, K. *A Study of Software Metrics for Predicting Software Errors.* M.S. Thesis, University of Wisconsin, Milwaukee, Wisconsin, Dec 1986.

[A34] SCHNEIDER, V. Prediction of Software Effort and Project Duration. *SIGPLAN Notices*, vol 14, no 6, Jun 1978, pp 49-59.

[A35] SHEN, V. Y., CONTE, S. D., and DUNSMORE, H. E. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Transactions on Software Engineering*, vol SE-9, no 2, Mar 1983, pp 155-165.

A15. Graph-Theoretic Complexity for Architecture

A15.1 Application. Complexity measures can be applied early in the product life cycle for development trade-offs as well as to assure system and module comprehensibility adequate for correct and efficient maintenance. Many system faults are introduced in the operational phase by modifications to systems that are reliable but difficult

to understand. In time, a system's entropy increases, making a fault insertion more likely with each new change. Through complexity measures the developer plans ahead for correct change by establishing initial order, and thereby improves the continuing reliability of the system throughout its operational life.

There are three graph-theoretic complexity measures for software architecture:

Static complexity is a measure of software architecture, as represented by a network of modules, useful for design tradeoff analyses. Network complexity is a function based on the countable properties of the modules (nodes) and inter-module connections (edges) in the network.

Generalized static complexity is a measure of software architecture, as represented by a network of modules and the resources used. Since resources are acquired or released when programs are invoked in other modules, it is desirable to measure the complexity associated with allocation of those resources in addition to the basic (static) network complexity.

Dynamic complexity is a measure of software architecture as represented by a network of modules during execution, rather than at rest, as is the case for the static measures. For example, modules may execute at different frequencies.

A15.2 Primitives

K = Number of resources, indexed by k = 1, ..., K

E = Number of edges, indexed by i = 1,..., E

N = Number of nodes, indexed by j = 1,..., N

c_i = complexity for program invocation and return along each edge e_i as determined by the user (such as operating system complexity).

Resource status array $R(K,E)$

$$r_{ki} = \begin{cases} 1 \text{ if } k^{th} \text{ resource is required for the } i\text{th edge } (e_i) \\ 0 \text{ otherwise} \end{cases}$$

d_k = complexity for allocation of resource k as determined by the user (eg, complexity associated with a procedure used to gain exclusive access to common data)

A15.3 Implementation. Using nodes and edges, a strongly connected graph of the network is required. A strongly connected graph is one in which a node is reachable from any other node. This is accomplished by adding an edge between the exit node and the entry node. Each node represents a module that may or may not be executed concurrently with another module. Each edge represents program invocation and return between modules. In this case the edges are called single paths.

A15.3.1 Static Complexity. Once a strongly connected graph is constructed, with modules as nodes, and transfer of control as edges, the static complexity is calculated as

$$C = E - N + 1$$

A15.3.2 Generalized Static Complexity. Resources (storage, time, logic complexity or other measurable factors) are allocated when programs are invoked in other modules. Given a network and resources to be controlled in the network, the generalized static complexity associated with allocation of these resources is

$$C = \sum_{i=1}^{E} \left(c_i + \sum_{k=1}^{K} (d_k * r_{ki}) \right)$$

A15.3.3 Dynamic Complexity. A change in the number of edges may result from module interruption due to invocations and returns. An average dynamic network complexity can be derived over a given time period to account for the execution of modules at different frequencies and also for module interruption during execution. Dynamic complexity is calculated using the formula for static complexity at various points in time. The behavior of the measure (except time average) is then used to indicate the evolution of the complexity of the software.

A15.4 Interpretation. For a typical subsystem, 10 represents the maximum ideal static complexity. A very complex system may be composed of a number of such subsystems. When the computed complexity is too large, one can consider more use of pipelining, fewer modules (if feasible), or more subsystems.

For a typical subsystem, $20E$ (E = number of edges) represents maximum ideal generalized static complexity, if the complexity of program invocation is 20 and no resources are needed.

The dynamic complexity is generally higher than the static complexity. Ideally, the dynamic complexity should be as close to the static complexity as possible. Addition or deletion of modules during execution affects the dynamic complexity. Minimization of the frequency of module interruption during execution reduces average dynamic network complexity. This results in an overall improvement in software reliability.

A15.5 Considerations. Pipeline architectures have lower static complexity value, since there is less overhead for transfer of control than for hierarchies. Decisions can be made on whether to implement services as distinct modules by combining the result obtained with this measure with the internal module complexity measures. For large systems, automation is desirable.

When the generalized static complexity is too large, one can consider more use of pipelining, fewer modules (if feasible), more subsystems, use of fewer resources, or less costly methods of resource acquisition.

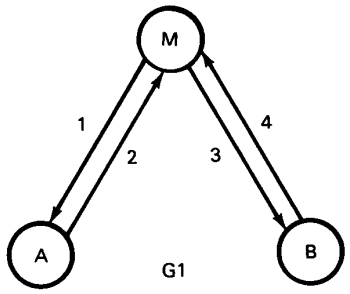
When the dynamic network complexity is too large, one should consider reducing it by optimizing the dynamic creation and deletion of modules, or by minimizing task interruptions by timing adjustments, or both.

A15.6 Training. Training in graph theory is recommended for manual application of the measure. Training is also needed to interpret the results for suitable design trade-off decisions for software architecture design.

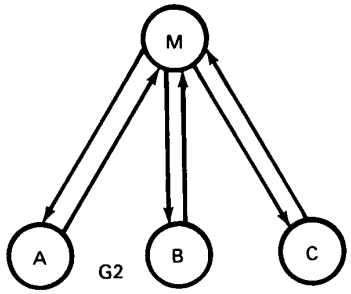
A15.7 Example. Static complexity measures for network hierarchies and pipelines are shown in Figs A15.7-1 and A15.7-2. Detailed examples for dynamic creation/deletion of modules, module interruption, modules in execution at different frequencies, and dynamic network complexity are illustrated in Halstead [A30].

A15.8 Benefits. These complexity measures, applied to the software architecture, can aid in identifying networks that are overly complex, and that are likely to result in high system overhead,

Fig A15.7-1
Hierarchy Network

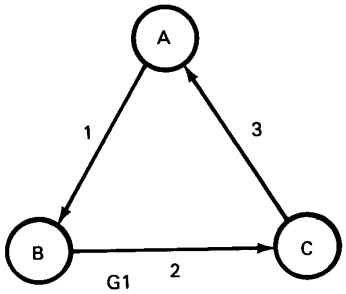


# Edges = 4
# Nodes = 3
C = 4 - 3 + 1 = 2

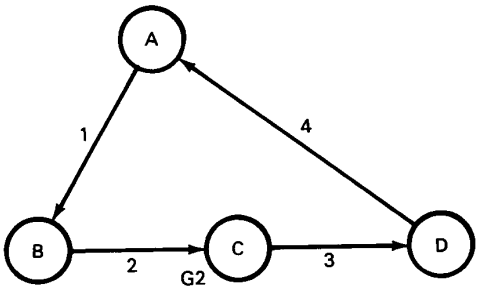


# Edges = 6
# Nodes = 4
C = 6 - 4 + 1 = 3

Fig A15.7-2
Pipeline Networks



# Edges = 3
# Nodes = 3
C = 3 - 3 + 1 = 1



# Edges = 4
# Nodes = 4
C = 4 - 4 + 1 = 1

difficulty in testing, and difficulty in meeting performance requirements. They can also aid in identifying areas that make excessive use of resources, and areas in which resource acquisition overhead is high.

A15.9 Experience. The static complexity measure has been used extensively for design and code [A36]. Generalized static and dynamic complexity measures have been used in experiments, [A38], but not operationally.

A15.10 References. See also Conte, Dunsmore, and Shen [A28] and Halstead [A30].

[A36] HALL, N. R. *Complexity Measures for Systems Design.* Ph.D. Dissertation, Polytechnic Institute of New York, Brooklyn, Jun 1983.

[A37] HALL, N. R. and PREISER, S. Dynamic Complexity Measures for Software Design. *Proceedings of Total Systems Reliability Symposium,* IEEE Computer Society, Dec 1983.

[A38] HALL, N. R. and PREISER, S. Combined Network Complexity Measures. *IBM Journal of Research and Development,* Jan 1984.

[A39] McCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering,* vol SE-2, no 4, Dec 1976, pp 308-320.

A16. Cyclomatic Complexity

A16.1 Application. This measure may be used to determine the structural complexity of a coded module. The use of this measure is designed to limit the complexity of a module, thereby promoting understandability of the module.

A16.2 Primitives

N = Number of nodes (sequential groups of program statements).
E = Number of edges (program flows between nodes).
SN = Number of splitting nodes (nodes with more than one edge emanating from it).
RG = Number of regions (areas bounded by edges with no edges crossing).

A16.3 Implementation. Using regions, or nodes and edges, a strongly connected graph of the module is required. A strongly connected graph is one in which a node is reachable from any other node. This is accomplished by adding an edge between the exit node and the entry node. Once the graph is constructed, the measure is computed as follows:

$C = E - N + 1$

The cyclomatic complexity is also equivalent to the number of regions (RG) or the number of splitting nodes plus one (SN+1). If a program contains an N-way predicate, such as a CASE statement with N cases, the N-way predicate contributes N-1 to the count of SN.

A16.4 Interpretation. For a typical module, ten represents a maximum ideal complexity. For a very complex system, the cyclomatic number (complexity) may be larger. If the calculated complexity number is too large, modularization of the code is required [A41].

A16.5 Considerations. Since the construction of the graph can be a tedious, error-prone process, automation of graph construction is desirable. The cyclomatic complexity method does not take into account the complexity of sequential program statements, a single program statement, or control flow caused by a multi-branching conditional statement.

The cyclomatic complexity can be used to limit intra-module complexity. However, to accomplish this one may increase the intermodule complexity by adding to the number of modules needed to accomplish a given function. The user should optimize both inter- and intra-module complexity, not one at the expense of the other.

A16.6 Training. If a tool is used, training in the use of a tool is required. If performed manually, training in graph construction is required.

A16.7 Example. Figure A16.7-1 illustrates a strongly connected graph. The nodes are labeled A (the entry node) through J (the exit node). Each node represents a block of code where control flow can only go from one statement to the next. There are 14 edges, 5 numbered regions, and 10 nodes. The complexity can be computed using either of the three methods:

C = complexity = number of regions = 5

The five regions are bounded by the paths:

A-B-C-E-G-J-A and J-A
A-B-C-E-F-I-J-A and A-B-C-E-G-J-A
A-B-C-E-F-H-J and A-B-C-E-F-I-J
A-B-C-D-J and A-B-C-E-F-H-J
D-D (this could be a DO statement)

Nodes and edges: complexity = $E - N + 1$
$= 14 - 10 + 1 = 5$

Splitting nodes: complexity = $SN + 1 = 4 + 1 = 5$

The four splitting nodes are: C, D, E and F

A16.8 Benefits. The cyclomatic number can be a good indicator of module complexity, and may be used to keep modules within an understandable and workable size. This measure also results in the computation of the number of distinct paths in a module without considering the complexity of an individual statement.

A16.9 Experience. There is empirical evidence [A41] that the cyclomatic complexity can reduce the difficulty encountered by programmers. There is some evidence that using the cyclomatic measure results in a module with fewer faults. Basili and Perricone [A40] provide a critical evaluation of this measure.

A16.10 References. See also Conte, Dunsmore, and Shen [A28] and McCabe [A39].

[A40] BASILI, V. R., and PERRICONE, B. T. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Jan 1984.

[A41] McCABE, T. J. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.* National Bureau of Standards Special Publication 500-99, Dec 1982.

A17. Minimal Unit Test Case Determination

A17.1 Application. This measure determines the number of independent paths through a module so that a minimal number of covering test cases can be generated for unit test. It is applied during unit testing.

A17.2 Primitives

N = number of nodes; a sequential group of program statements.

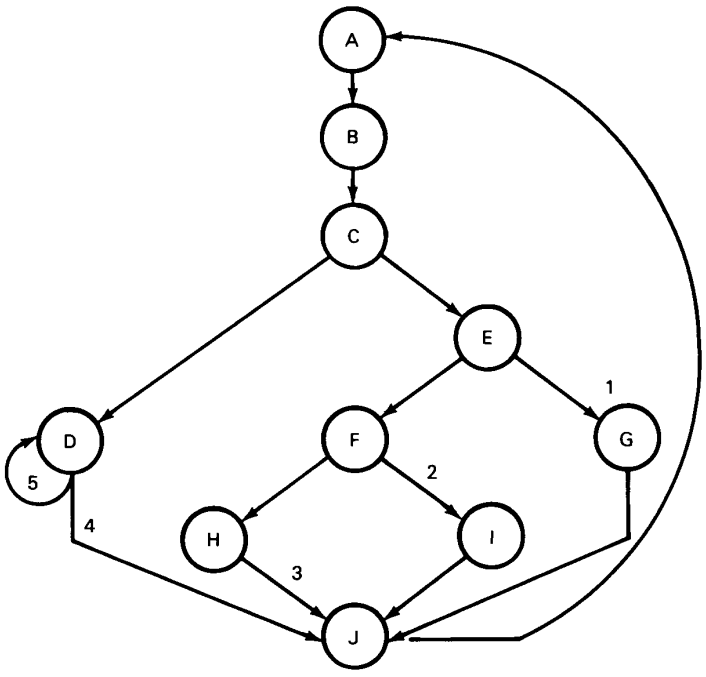
E = number of edges; program flow between nodes.

SN = number of splitting nodes; a node with more than one edge emanating from it.

RG = number of regions; in a graph with no edges crossing, an area bounded by edges.

A17.3 Implementation. The cyclomatic complexity is first computed using the cyclomatic

Fig A16.7-1
Module Control Graph



complexity measure described in A16. The complexity of the module establishes the number of distinct paths. The user constructs test cases along each path so all edges of the graph are traversed. This set of test cases forms a minimal set of cases that covers all paths through the module.

A17.4 Interpretation. The minimal set of test cases is the number of independent paths through the module. Execution of these test cases generally causes every statement and decision outcome to be executed at least once. Additional test cases may be added (eg, to test upper and lower bounds on variables).

A17.5 Considerations. The construction of the graph can be a tedious, error-prone process. Thus, automation of the graph construction is desirable. Once the graph is constructed, the generation of test cases is done by hand. Since the graph produces a minimal set of test cases, additional test cases should be generated to comprehensively test the module.

A17.6 Training. Training may be required in graph construction and interpretation in order to determine the minimal set of test cases.

A17.7 Example. Figure A17.7-1 illustrates a strongly connected graph. The nodes are labeled A (entry node) through J (exit node). Each node represents a block of code where control flow can only go from one statement to the next. Using the method for computing complexity (A16), the complexity of the module represented in Fig A17.7-1 is five. There are five distinct paths through the graph. They are:

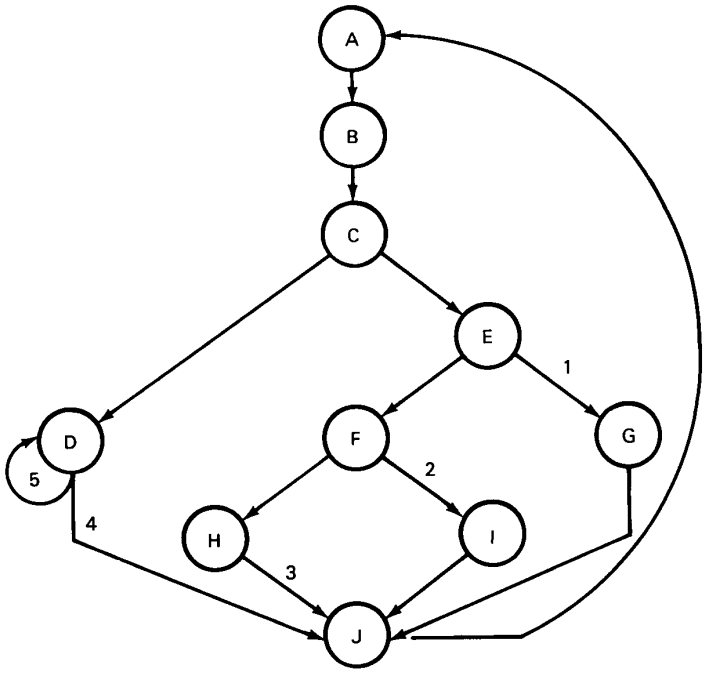
A-B-C-E-G-J
A-B-C-E-F-I-J
A-B-C-E-F-H-J
A-B-C-D-J
A-B-C-D-D-J

The user selects five test cases causing the five distinct paths to be executed. The selection of minimal test case is demonstrated in McCabe [A41].

A17.8 Benefits. The minimal test case provides a rigorous method of determining minimal test coverage. This same set can be used, with appropriate modifications, during maintenance.

A17.9 Experience. There is empirical evidence [A41] that determining minimal test cases reduces

Fig A17.7-1
Module Control Graph



the difficulty encountered by programmers in testing every path in a module. The advantage of this approach is the completeness of the test of a module and the assurance that all segments of the module are tested.

A17.10 References. See Conte, Dunsmore, and Shen [A28] and McCabe [A39, A41].

A18. Run Reliability

Run reliability (R_k) is the probability that k randomly selected runs (corresponding to a specified period of time) will produce correct results.

A18.1 Application. The run reliability measure may be applied at any phase in the life cycle when the set of possible discrete input patterns and states can be identified. The sequence is randomly selected based on the probability assigned to selection of any one of them.

A18.2 Primitives

NR = Number of runs in a given test sample.

nc = Number of correct runs in a given test sample.

k = Number of runs for the specified period of time.

S = Sample space of possible input patterns and states. (S_i, i =1 ...) are elements in the sample space. For each input pattern and state there is a unique output pattern and state. Thus, S_i designates a given ((input, state), (output, state)) that constitutes a run.

P = Probability measure over the sample space.

P_i = Probability that the ith run is selected from the sample space.

A18.3 Implementation

A18.3.1 Sample Space (S). The sample space is viewed as the set of possible inputs into the program. In the early phases of the life cycle, the sample space consists of operational scenarios. For instance, in the early requirements phase the inputs and states are the listed requirements for the program. In the later phases of the life cycle, the sample space consists of all detailed input and state conditions of the program. For a given life cycle phase, a single sample space (S) is determined and a probability measure (P) is assigned. Runs are then generated by randomly choosing input patterns and states from the sample space according to the assigned probability P.

A18.3.2 Probability Measure (P). For each possible run i, a number p_i (between zero and one inclusive) must be assigned so that the sum of all the p_i's is 1. The simplest case is the assumption of a uniform probability distribution. However, in the operational environment some runs are more likely to occur than others, and should be weighted appropriately.

A18.3.3 Test NR Randomly Selected Runs. The accuracy of the measure R_k is determined by the number of runs NR used for estimation. The runs are randomly selected according to the probability distribution. The results of each run are examined for correctness. The number of correct runs is nc.

A18.3.4 Probability of a Successful Run (P_r). The probability of a randomly selected run being correct (P_r) is estimated by the sum of the probabilities for the correct runs, divided by the sum of the probabilities for all runs. In the case of a uniform distribution $P_r = nc/NR$.

A18.3.5 Run Reliability (R_k). Given k randomly selected runs during the specified period of time, the probability that all k runs give correct results is $R_k = P_r^k$.

A18.4 Interpretation. The reliability obtained using the number of runs for the specified period of time is the probability that the software will perform its intended function for the specified period of time under the stated conditions. Thus, it satisfies the standard definition for systems reliability [A42, A46] and may be combined with other components of the system, such as the hardware component, using the usual rules of probability.

A18.5 Considerations. The run reliability is not defined until the sample space and probability measure are defined for the specific application. These are not unique for the program. They must be carefully constructed by all parties with knowledge of the application environment, the program, and the test procedures, as well as the construction of probability spaces. Construction of the input sample space (A18.3.1) can be done in a number of different ways. There must be a unique output for each input. Thus attention must be given to the initialization of the variables. If more than one input channel is possible, an input pattern may be a vector over time. If a real time system is involved, the exact time of each input must be specified. When a hardware/software system is considered, there must be agreement on the accuracy of the input magnitude and timing.

During the specification or early design phase, a sample space might involve the variety of application scenarios using the various system functions in different ways. A run might be judged successful if all the functions needed in that scenario are appropriately specified.

Construction of the probability measure (P) is possibly the most difficult and potentially misleading error source. The uniform distribution on all possible inputs is the simplest and might be agreed on for test purposes, but may not be representative of the application environment. An inappropriate probability measure will not predict what happens in the field. Changes in the way in which the system is used constitute a change in the probability measure and may change the actual and predicted run reliability. A high reliability measure during one phase or period of time will be applicable to other stages only to the extent that the program remains constant and the sample space and probability measure is representative of the runs that will be made in that other phase.

A18.6 Training. Training should include the basic four chapters in Feller [A44], random sampling techniques, a beginning level of statistical knowledge, and an understanding of the application environment.

A18.7 Example

A18.7.1 Concept Phase. In order for the final system to work reliably, a list of necessary functions must be identified. Suppose a list of functions has been prepared which contains $NR = 20$ different scenarios. The list of functions and scenarios are given to a review team that traces each of the scenarios to check that all the functions required for mission success are available. Further, suppose the review team reports that in one of the scenarios a function necessary for the success of the mission was not in the list. The estimate of the system reliability at this point is .95. After adding the necessary function, one has a new system that works for these scenarios. In order to make a statement concerning the reliability of the new system, however, one would need to "randomly" select a new set of scenarios and test again. Suppose then, in this second test, that in all 20 scenarios the mission could have been performed successfully in the sense that all necessary functions were in the list. This does not mean, however, that following the next scenario through will necessarily find satisfactory results. Suppose the required reliability for the run is .9 and we wish to demonstrate that this requirement has been met with an accepted 5% risk of failing to reject an unreliable program. That is to say, we wish to test the null hypothesis that the probability of a successful run is less than .9 at the 5% significance level. The probability of making 30 successful runs when the probability of success is less than or equal to .9 is at most $.9^{30} = .047$. Thus, by obtaining success in 30 randomly selected test cases, one could conclude that the requirement of a run reliability of .9 had been met with less than a 5% chance of making an incorrect conclusion.

A18.7.2 Implementation Phase. Suppose the software program has nautical miles (nmi) as input and the range is broken up into the following categories: (0, 500 nmi), (500, 1000 nmi), (1000, 2000 nmi), and (above 2000 nmi). In addition, suppose the associated probabilities assigned by the user are ¼, ⅛, ⅛, and ½ respectively. The user believes that ¼ of the operational time ranges will come from the interval (0, 500), ⅛ of the time from (500, 1000), etc. A random selection of ranges are generated according to the distribution where, once an interval is chosen, a range is drawn uniformly from the chosen interval. Now suppose nr_i ranges were randomly selected from interval i with nc_i being run successfully, $i=1, 2, 3, 4$. The estimated reliability is then

$$R = \frac{nc_1}{4nr_1} + \frac{nc_2}{8nr_2} + \frac{nc_3}{8nr_3} + \frac{nc_4}{2nr_4}$$

Another example within the implementation phase is as follows:

Alphameric input patterns can be defined for use in code walk-through or computer runs. Consider a module with two input variables A and B. The range as viewed by the user for A is the alphameric characters A through Z and for B is all positive real numbers. Regardless of how the user views the ranges, an 8 bit computer will view each input as one of 2^8 possibilities. The two characters together will be one of 2^{16} possibilities, presuming here that timing is of no concern. For simplicity it might be agreed that each of the 2^{16}th possible inputs representing an input for which the algorithm was intended will have a constant probability p of being chosen, and the rest will have probability p'. One can easily develop a random number generator to randomly select among these possible inputs. One can either select from both intended and unintended inputs and weight the results, or one can develop a random number generator that will have an appropriate

probability for selecting from the intended or unintended inputs. Numerical results for real numbers will not always be exactly correct, but are said to be "correct" if the difference is within some acceptable bounds of accuracy. The results computed by the computer operating on these inputs may then be compared to those calculated by hand or other "oracle", perhaps a program in double precision instead of single precision.

A18.8 Benefits. The technique is solidly based theoretically and very easy to interpret if one accepts the assumed sample space and probability measure. This measure can be integrated with a hardware reliability measure to form system reliability, maintainability, and availability estimates.

A18.9 Experience. The concepts and experiences with this measure are documented in [A43, A45, and A47-A49].

A18.10 References

[A42] ANSI/IEEE 729-1983, IEEE Standard Glossary of Software Engineering Terminology.

[A43] BROWN, J. R. and LIPOW, M. Testing for Software Reliability. Proceedings of the 1975 International Conference on Reliable Software, Apr 21-23, 1975, IEEE catalog no 75-CH0940-7CSR, pp 518-527.

[A44] FELLER, W. *An Introduction to Probability Theory and Its Applications.* New York: John Wiley and Sons, Inc, 1957, chaps I, V, VI, and IX.

[A45] LIPOW, M. *Measurement of Computer Software Reliability.* AFSC-TR-75-05, 1979.

[A46] MIL Std 721.

[A47] NELSON, E. C. *A Statistical Basis for Software Reliability Assessment.* TRW-SS-73-03, Redondo Beach, California: TRW, Mar 1973.

[A48] NELSON, E. C. *Software Reliability.* TRW-SS-75-05, Redondo Beach, California: TRW, Nov 1975.

[A49] THAYER, T. A., LIPOW, M and NELSON, E. C. *Software Reliability: A Study of Large Project Reality.* New York: North Holland Publishing Co, 1978.

A19. Design Structure

A19.1 Application. This measure is used to determine the simplicity of the detailed design of a software program. Also, the values determined for the associated primitives can be used to identify problem areas within the software design.

A19.2 Primitives

$P1$ = total number of modules in the program
$P2$ = number of modules dependent on the input or output
$P3$ = number of modules dependent on prior processing (state)
$P4$ = number of database elements
$P5$ = number of non-unique database elements
$P6$ = number of database segments (partition of the state)
$P7$ = number of modules not single entrance/single exit.

A19.3 Implementation. The design structure measure is the weighted sum of six derivatives determined by using the primitives given above. The six derivatives are:

D_1 = design organized top down (Boolean)
D_2 = module dependence ($P2/P1$)
D_3 = module dependent on prior processing ($P3/P1$)
D_4 = database size ($P5/P4$)
D_5 = database compartmentalization ($P6/P4$)
D_6 = module single entrance single exit ($P7/P1$).

The design structure measure (DSM) can be expressed as:

$$DSM = \sum_{i=1}^{6} W_i D_i$$

The weights (W_i) are assigned by the user based on the priority of each associated derivative. Each W_i has a value between 0 and 1 ($\Sigma_i W_i = 1.0$).

A19.4 Interpretation. The value of the design structure measure is scaled between 0 and 1 by the appropriate weights. A score near 0 is considered better than a score near 1. Design structure measure values that are considered high could be traced to "offending" primitive(s) to highlight particular priority areas for change in the design. As changes are made to the design structure, the incremental design structure measure values can be plotted to show whether improvements are being made and how rapidly. Also, comparisons of design structure measures between different projects allows a relative scaling to be made for values determined.

A19.5 Considerations. Each primitive is calculated to help identify those characteristics in the

design that promote simplicity in the construction of modules and databases.

A19.6 Training. Training would be necessary to understand the definitions of the primitives for the particular application. The use of a program design language (PDL) would enhance and standardize the definition procedure for primitives.

A19.7 Example. Previous to calculating the primitive (and associated derivative) values, the following steps should occur:

(1) The definitions of the primitives for the particular application must be determined. For example, if a program design language (PDL) were being used to develop the software design, then the primitives would be defined in association with the PDL constructs.

(2) Priorities should be determined for each derivative (D_i). These priorities should then be used to assign the associated weighting factors (W_i). If no priorities are established, the weight for each derivative would be one-sixth (⅙).

To arrive at a value for the design structure measure, assume that the primitives have the following values:

$P1 = 250$ $P3 = 50$ $P5 = 500$ $P7 = 10$
$P2 = 125$ $P4 = 2500$ $P6 = 50$

The derivative values are calculated to be:

$D_1 = 0$ (assumed, top-down design)
$D_2 = .5\ (P2/P1)$
$D_3 = .2\ (P3/P1)$
$D_4 = .2\ (P5/P4)$
$D_5 = .02\ (P6/P4)$
$D_6 = .04\ (P7/P1)$

Using a weight of one-sixth (⅙) for each W_i the design structure measure value is:

$$DSM = \sum_{i=1}^{6} W_i D_i$$

$$= \tfrac{1}{6}(0) + \tfrac{1}{6}(.5) + \tfrac{1}{6}(.2) + \tfrac{1}{6}(.2) + \tfrac{1}{6}(.02) + \tfrac{1}{6}(.04)$$

$$= .16$$

A19.8 Benefits. The design structure measure will give an indication of the simplicity (value near 0) or complexity (value near 1) of the software design.

A19.9 Experience. There is limited published experience with the use of the design structure measure. This measure is similar to the completeness measure [A35]. The experience consists of the determination of the proper definitions and measurements of primitives and calculation of associated derivatives. This technique worked successfully for the completeness measure, and similar results are expected for the design structure measure.

A19.10 References

[A50] FRANCIS, J. et al. *The Application of Software Quality Metrics to Mx Operational G&C Software.* Dynamics Research Corporation, Sept 1982.

[A51] McCALL, J., RICHARDS, P., and WALTERS, G. *Factors in Software Quality*, RADC-TR-77-369, 3 vols, Nov 1977.

[A52] McCALL, J., and MATSUMOTO, M. *Metrics Enhancements.* RADC-TR-80-109, 2 vols, Apr 1980.

[A53] MILLER, D. et al. *The Application of Software Quality Metrics to PCKR Operational Guidance and Control Software.* Dynamics Research Corporation, Sept 1983.

A20. Mean Time to Discover the Next *K* Faults

A20.1 Application. This measure can be used to project the average length of time until the next K faults are discovered. This can provide an indication of how much time would be required until a desired level of reliability is achieved.

A20.2 Primitives

f = number of failures found from the beginning of testing to the present

t_i = observed time between the $(i-1)$st and ith failure for a given severity level $i=1,\ldots$

A20.3 Implementation. The mean time to failure (see A30) should be used to estimate the mean time to discover the next K faults in the software. This can then be expressed as:

$$\text{mean time to discover the next } K \text{ faults} = \sum_{i=f}^{f+K-1} \widehat{\text{MTTF}}_i$$

where $\widehat{\text{MTTF}}_i$ is an estimate of the mean time to failure between the ith and $(i+1)$st failure.

The estimate of the mean time to failure between the ith and $(i+1)$st failure can be calculated using any of the software models based upon time between failures [A56-A58]. Suppose that estimate is:

$$\widehat{\text{MTTF}}_i = \hat{E}\{\text{time between the } i\text{th and the } (i+1)\text{st failures}\}\ f \le i \le f+K-1$$

Notice that if K=1, then we simply have the estimate of the MTFF (see A30) between the fth and (f+1)st failure. Then the estimate is:

$$\begin{pmatrix}\text{mean time to}\\ \text{discover the}\\ \text{next } K \text{ faults}\end{pmatrix} = \sum_{i=f}^{f-K+1} \hat{E} \begin{Bmatrix}\text{time between}\\ \text{the } i\text{th and the}\\ (i+1)\text{st}\end{Bmatrix}$$

By estimating the mean time to failure, one can then estimate how much additional time will be required to discover all, or a subset, of the estimated remaining faults. One can judge how long it will take before the software achieves a desired level of reliability.

A20.4 Interpretation. Actual numerical calculations of the estimate depends on the form of the model chosen to represent the distribution of MTTF. Within the software life cycle, the estimate is useful when planning for time necessary to achieve stabilization.

A20.5 Considerations. To apply this measure, one must collect the times between failure and then fit the data to one of the software models. For better estimates, it is desirable to have time units with as much resolution as feasible, eg, CPU execution time is especially appropriate in a software development environment. Sometimes wall-clock time is the only time available, and in this situation estimates can be made, but with much less precision. For the situations where operational or application usage requires less resolution, estimates based on wall-clock time units would be appropriate.

During the software development life cycle, evaluation of model predictions versus actual time required to achieve a certain level of reliability should be used to construct realistic schedules.

A20.6 Training. The implementation requires knowledge of the application of software models that use the time between failure data.

A20.7 Example. In the fitting of the Jelinski and Moranda "de-eutrophication" model [A56], the estimate of the mean time to failure between the ith and (i+1)st error is

$$\text{MTTF}_i = \hat{E}\{\text{time between } i\text{th, } (i+1)\text{st})\} = \frac{1}{\hat{Q}(\hat{NF}-i)}$$

where $\hat{Q}$ is the estimate of the proportionality constant in that model and $\hat{NF}$ is the estimate of the total number of faults in the program. The estimate of the mean time to discover the next K faults is then:

$$\sum_{i=f}^{f+K-1} \frac{1}{\hat{Q}(\hat{NF}-i)}$$

For illustrative purposes suppose f=20 and $\hat{Q}$ and $\hat{NF}$ were calculated as .05 and 100 respectively. If one were interested in the mean time to discover the next K=2 failures then the estimate becomes:

$$\sum_{i=20}^{21} \frac{1}{(.05)(100-i)} = \frac{1}{(.05)(80)} + \frac{1}{(.05)(79)} = 0.50$$

A20.8 Benefits. This measure can be plotted against K (K=1, 2, 3,...NF remaining number of faults) to judge the integrity of the system at any point in time.

A20.9 Experience. Experience is a function of application of any of the given models in the references.

A20.10 References. See also Musa, Iannino, and Okumoto [A7].

[A54] GOEL, AMRIT L., and OKUMOTO, K. *A Time Dependent Error Detection Rate Model for Software Performance Assessment with Applications.* RADC #F3062-78-C-0351, Mar 1980, p 76.

[A55] LITTLEWOOD, B. A Stochastic Reliability Growth: A Model for Fault Removal in Computer Programs and Hardware Designs. *IEEE Transactions on Reliability*, vol R-30, no 4, Oct 1981, pp 313-320.

[A56] MORANDA, PAUL L., and JELINSKI, Z. *Final Report on Software Reliability Study.* McDonnell-Douglas Astronautics Company, MDC Report no 63921, 1972.

[A57] MUSA, J. D. A Theory of Software Reliability and Its Application. *IEEE Transactions on Software Engineering*, vol SE-1, no 3, 1975, pp 312-327.

[A58] SCHICK, GEORGE J., and WOLVERTON, RAY, W. An Analysis of Competing Software Reliability Models. *IEEE Transactions on Software Engineering*, vol SE-4, no 2, 1978, pp 104-120.

A21. Software Purity Level

A21.1 Application. The software purity level provides an estimate of the relative fault-freeness of a computer program at any specified point in time during the operational phase.

A21.2 Primitives

t_i = observed times between failures (eg, execution time) for a given severity level

f = total number of failures in a given time interval

A21.3 Implementation. Suppose f failures have occurred during the operational phase of the life cycle of the program. Suppose also the length of time in this phase when the fth failure is detected is t_f. Then the estimate of the purity level of the program at that point in time is

$$PL = \frac{\hat{Z}(t_0) - \hat{Z}(t_f)}{\hat{Z}(t_0)}$$

where t_0 is the start of the operations and maintenance phase and $\hat{Z}(t)$ is the estimated hazard (or failure) rate at time t. The hazard rate is defined as the conditional probability that a software failure happens in an interval of time (t, $t+\Delta t$) given that the software has not failed up to time t. It is derived from any of the software reliability models that use time between failures data t_i. *PL* denotes the relative change in the hazard rate function from the beginning of the operations and maintenance phase to the time of the fth failure detection.

A21.4 Interpretation. The closer *PL* gets to 1 the greater the reliability of the program. Values close to 1 indicate that many of the initial faults in the program have been eliminated.

A21.5 Considerations. The application of this measure is especially applicable for those classes of models in which the total number of remaining faults in the program cannot be predicted (eg, Moranda's geometric model [A60 and A61]). Its use is restricted to those phases where significant test results can be obtained.

A21.6 Training. Familiarity with the various software reliability models employing time between failure data is essential.

A21.7 Example. For the Moranda geometric model (A60), the estimate of the hazard rate at any time t between the occurrence of the ith failure and the $(i+1)$st is:

$$\hat{Z}(t) = \hat{D}\hat{Q}^i$$

$\hat{D}$ is the estimate of the initial hazard rate at time t_o and $(0<Q<1)$ is the estimate of the proportionality constant in the model [A60]. Then the purity level using this model is

$$PL = \frac{\hat{Z}(t_0) - \hat{Z}(t_f)}{\hat{Z}(t_0)} = 1 - \hat{Q}^f$$

A21.8 Benefits. A plot of *PL* versus n (the number of failures) can be used to see the relative "fault-freeness" of the program as the number of detected faults increases.

A21.9 Experience. Limited experience is provided in the references.

A21.10 References. See also Musa, Iannino, and Okumoto [A7].

[A59] FARR, W. H. *A Survey of Software Reliability, Modeling and Estimation.* Dahlgren, Virginia: Naval Surface Weapons Center, NSWC TR 82-171, Sept 1983.

[A60] MORANDA, PAUL Predictions of Software Reliability During Debugging. *1975 Proceedings of the Annual Reliability and Maintainability Symposium*, 1975.

[A61] MORANDA, PAUL Event-Alerted Rate Models for General Reliability Analysis. *IEEE Transactions on Reliability*, vol R-28, no 5, 1979, pp 376-381.

A22. Estimated Number of Faults Remaining (by Seeding)

A22.1 Application. The estimated number of faults remaining in a program is related to the reliability of the program. There are many sampling techniques that estimate this number. This section describes a simple form of seeding that assumes a homogeneous distribution of a representative class of faults [A67].

This measure can be applied to any phase of the software life cycle. The search for faults continues for a determined period of time that may be less than that required to find all seeded faults. The measure is not computed unless some non-seeded faults are found.

A22.2 Primitives

N_s = the number of seeded faults

n_s = the number of seeded faults found

n_F = the number of faults found that were not intentionally seeded

A22.3 Implementation. A monitor is responsible for error seeding. The monitor inserts (seeds) N_s faults representative of the expected indigenous faults. The test team reports to the monitor

the faults found during a test period of predetermined length.

Before seeding, a fault analysis is needed to determine the types of faults and their relative frequency of occurrence expected under a particular set of software development conditions [A49, A65]. Although an estimate of the number of faults remaining can be made on the basis of very few inserted faults, the accuracy of the estimate (and hence the confidence in it) increases as the number of seeded faults increases.

Faults should be inserted randomly throughout the software. Personnel inserting the faults should be different and independent of those persons later searching for the faults. The process of searching for the faults should be carried out without knowledge of the inserted faults. The search should be performed for a previously determined period of time (or effort) and each fault reported to the central monitor.

Each reported fault should be reviewed to determine whether it is in the class of faults being studied, and if so, whether it is a seeded or an indigenous fault. The maximum likelihood estimate of the number of indigenous (unseeded) faults in the specified class is:

$$\hat{NF} = \frac{n_F N_s}{n_s} \qquad \text{(eq A22-1)}$$

where $\hat{NF}$ is truncated to the integer value. The estimate of the remaining number of faults is then: $\hat{NF}_{rem} = \hat{NF} - n_F$.

The probability of finding n_F of NF indigenous faults and n_F of N_s seeded faults, given that there are $n_F + n_s$ faults found in the program is $C(N_s,n_s)\,C(NF,n_F) / C(NF+N_s, n_F+n_s)$ where the function $C(x,y) = x!/(x-y)!y!$ is the combination of "x" things taken "y" at a time. Using this relation one can calculate confidence intervals.

A22.4 Interpretation. The number of seeded and indigenous faults discovered enables the number of faults remaining to be estimated for the fault type being considered. The phrase "fault type being considered" means that if, for example, one was seeding typographical faults, one would not expect a good prediction of the number of omitted functions or algorithmic faults. If one were seeding faults of the type found in unit testing, one would not expect the estimate to apply to those interface faults found in integration testing.

A22.5 Considerations. This measure depends on the assumption that the seeded (inserted) faults are representative of existing (unseeded) faults. In general, not all faults are equally likely to be found in any given test procedure. To assure that the seeded faults are representative of a specific class of faults, and the result will have the desired accuracy, enough faults of the kind expected should be inserted to provide the desired accuracy.

Faults are corrected in the original software only after the fault has been determined to be unseeded by rerunning the test case on the original unseeded software. Good configuration management practices are needed to assure that the seeded version is not the version finally released and that indigenous faults discovered are corrected only in the unseeded version.

The correction of the indigenous faults in the original version raises the problem of whether the correction introduces further faults. At this time this must be left as an engineering judgment as to whether the new (corrected) program needs to be seeded to give an estimate of its reliability or whether the estimate of the faults remaining in the new (corrected) program is $NF - n_F$ where n_F is the number of indigenous faults found and corrected.

In lieu of actually seeding faults in a program, the number of "seeded" faults may sometimes be generated by examining the results of testing by two independent test teams [A71 and A63]. The seeded faults would be those faults identified by both test teams. The unseeded faults are those identified by either, but not both, of the test teams. An estimate of the number of remaining faults can then be calculated using Eq A22-1.

A22.6 Training. No additional training is required for the developers or the testers. Training in seeding the faults and monitoring the process is needed. Some general training and practice in classifying faults is also needed.

A22.7 Example. Suppose 100 were seeded. During the prescribed test period, 50 of the 100 seeded faults and 2 unseeded faults were found.

Then:

$$N_s = 100$$
$$n_s = 50$$
$$n_F = 2$$
$$\hat{NF} = \left\lfloor \frac{n_F N_s}{n_s} \right\rfloor = \left\lfloor \frac{2(100)}{50} \right\rfloor = 4$$

Thus, it is estimated that there are $\hat{NF}_{rem} = \hat{NF} - n_F = 2$ faults remaining.

A22.8 Benefits. This procedure encourages the reporting of faults and competition between developers and testers. Since faults are known to be there, the tester knows there is something to search for. The reporting of faults does not necessarily admit to an error on the part of the developers.

The procedure could be carried out by a customer monitor with little effort, high effectiveness, and minimal interference with the developer's normal patterns. For the cost of a second independent testing group, an estimate of the number of remaining faults can be obtained without seeding errors.

A22.9 Experience. Nippon Telephone and Telegraph reported [A73] good results applying their version of the seeding method to an operating system consisting of 150,000 lines of code. A mixture of seeding and a software reliability growth model at IBM Japan Ltd. for predicting the number of faults to be found during the operational phase is described. Techniques for seeding faults include: (1) deferring correction of actual faults by programmers; (2) mutants (eg, replacing "and" with "or") that are considered to be similar inherent defects due to misinterpretation of the specifications; and (3) transplanted defects (eg, adding or deleting a flow-control statement) simulating the design mistakes. In two experiments involving 3000 and 4000 lines of code for modules in an on-line terminal operating program, the authors claim to have predicted (to within $\pm 10\%$) the fraction of defects found during testing with those remaining to be found during operation. The authors claim that seeding is practical in their experience.

In the US, a number of published papers have developed the principles and techniques for seeding [A64, A65, A67, A70, A74, A75].

A22.10 References. See also Thayer, Lipow, and Nelson [A49].

[A62] BASIN, S. L. *Estimation of Software Error Rates Via Capture-Recapture Sampling.* Palo Alto: Science Applications Inc, Sept 1973.

[A63] BASIN, S. L. *Measuring the Error Content of Software.* Palo Alto: Science Applications Inc, Sept 1974.

[A64] BOWEN, J. B. and SAIB, S. H. *Association of Software Errors, Classifications to AED Language Elements.* Prepared under Contract Number NAS 2-10550 by Hughes Aircraft Company, Fullerton, and General Research Corporation, Santa Barbara, California, Nov 1980.

[A65] BOWEN, J. B. and SAIB, S. H. *Error Seeding Technique Specification.* Prepared under Contract Number NAS 2-10550 by Hughes Aircraft Company, Fullerton, and General Research Corporation, Santa Barbara, California, Dec 1980.

[A66] DACS. *Rudner Model, Seeding/Tagging.* Quantitative Software Models, DACS, SSR-1, Mar 1979, pp 3-56 to 3-58.

[A67] DURAN, J. W. and WIORKOWSKI, J. J. Capture-Recapture Sampling for Estimating Software Error Content. *IEEE Transactions on Software Engineering,* vol SE-7, Jan 1981.

[A68] FELLER, W. *An Introduction to Probability Theory and Its Applications.* New York: John Wiley and Sons, Inc, 1957, p 43.

[A69] *System Specification for Flexible Interconnect.* Specification Number SS078779400, May 15, 1980.

[A70] LIPOW, M. *Estimation of Software Package Residual Errors.* TRW Software Series SS-72-09, Redondo Beach: TRW E & D, 1972.

[A71] McCABE, THOMAS J. *Structural Testing.* Columbia, Maryland: McCabe & Associates, Inc, 1984.

[A72] MILLS, H. D. *On the Statistical Validation of Computer Programs.* FSC-72-6015, Gaithersburg, Maryland: Federal Systems Division, International Business Machines Corporation, 1972.

[A73] OHBA, M., et al. *S-Shape Reliability Control Curve: How Good Is It?* Proceedings COMPSAC82, IEEE Computer Society, 1982, pp 38-44.

[A74] RUDNER, B. Seeding/Tagging Estimation of Software Errors: Models and Estimates. RADC-TR-15, 1977.

[A75] SEBER, G. A. F. *Estimation of Animal Abundance and Related Parameters.* 2nd ed, New York: McMillan Publishing Co, 1982.

A23. Requirements Compliance

A23.1 Application. This analysis is used to verify requirements compliance by using system verification diagrams (SVDs), a logical interconnection of decomposition elements (eg, stimulus and response) that detect inconsistencies, incompleteness, and misinterpretations.

A23.2 Primitives

*DE*s = Decomposition elements
- Stimulus = External input
- Function = Defined input/output process
- Response = Result of the function
- Label = Numerical *DE* identifier
- Reference = Specification paragraph number

Requirement errors detected using SVDs:
- N_1 = Number due to inconsistencies
- N_2 = Number due to incompleteness
- N_3 = Number due to misinterpretation.

A23.3 Implementation. The implementation of a SVD is composed of the following phases:

(1) The decomposition phase is initiated by mapping the system requirement specifications into stimulus/response elements (*DE*s). That is, all key words, phrases, functional or performance requirements or both, and expected output are documented on decomposition forms (see Fig A23.3-1).

(2) The graph phase uses the *DE*s from the decomposition phase and logically connects them to form the SVD graph.

(3) The analysis phase examines the SVD from the graph phase by using connectivity and reachability matrices. The various requirement error types are determined by experiencing the system verification diagram and identifying errors as follows:

inconsistencies. Decomposition elements that do not accurately reflect the system requirement specification.

incompleteness. Decomposition elements that do not completely reflect the system requirement specification.

misinterpretation. Decomposition elements that do not correctly reflect the system requirement specification. These errors may occur during translation of the requirements into decomposition elements, constructing the SVD graph, or interpreting the connectivity and reachability matrices.

Fig A23.3-1
Decomposition Form

STIMULUS	LABEL	RESPONSE
	FUNCTION	
SPECIFICATION REFERENCE		

An analysis is also made of the percentages for the various requirement error types for the respective categories: inconsistencies, incompleteness, and misinterpretation.

Inconsistencies (%) = $N_1/(N_1+N_2+N_3) \times 100$
Incompleteness (%) = $N_2/(N_1+N_2+N_3) \times 100$
Misinterpretation (%) = $N_3/(N_1+N_2+N_3) \times 100$

This analysis can aid also in future software development efforts.

A23.4 Interpretation. The SVDs are analyzed by interpreting the connectivity and reachability matrices. The rules governing their use are as follows:

Connectivity Matrix. The rules governing the use of connectivity matrices are as follows:

Rule 1. The matrix size is $N \times N$ where N equals the number of decomposition elements in a given SVD.

Rule 2. Any (i, j) matrix element is equal to one, if DE_i transfers directly to DE_j. It is equal to zero otherwise.

Rule 3. Any matrix column of all zeros identifies a *DE* that is not transferred to or by any other *DE* and, therefore, must be an entry *DE*.

Rule 4. Any matrix row of all zeros identifies a *DE* that does not transfer to any other *DE* and, therefore, must be a terminal *DE*.

Rule 5. If there is a one along the diagonal, it indicates that a particular *DE* loops back to itself. Though possible, this is improbable at the requirement level and usually indicates a fault.

Rule 6. Any *DE* with an associated zero row and zero column indicates a stand alone system function. Though possible, they are rare and should be double checked for authenticity.

Reachability Matrix. The rules governing the use of reachability matrices include the rules 1-6 for connectivity matrices and the following reachability rule:

Rule 7. Any (i, j) matrix element is equal to one if DE_j can be reached (directly or indirectly) from DE_i. It is equal to zero otherwise.

An analysis of the percentages will aid requirements specifications and future code testing. The categories with the higher percentages should receive the most scrutiny for future requirement specifications and test case design.

A23.5 Considerations. SVDs can be implemented either manually or with the aid of automation. Automation is recommended for large SVDs. An SVD can be used during the system requirement definition phase and/or during the software requirement phase.

A23.6 Training. Training is required if automation is used. The manual approach requires a familiarization with logic methodology.

A23.7 Example. See Fig A23.7-1.

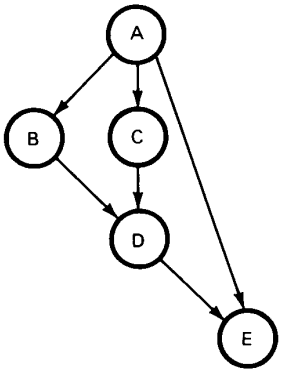
A23.8 Benefits. SVDs are good communication devices for internal and external reviews. The reachability matrix associated with SVDs can be used during the maintenance phase to assess modification impact and test direction. Requirements can be verified per life cycle phase.

A23.9 Experience. See also Fischer and Walker [A76].

A23.10 Reference

[A76] FISCHER, K. F. and WALKER, M. G. *Improved Software Reliability Through Requirement Verification.* IEEE Transactions on Reliability, vol R-28, no 3, Aug 1979, pp 233-239.

Fig A23.7-1



CONNECTIVITY ENTRY

	A	B	C	D	E
A	0	1	1	0	1
B	0	0	0	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

REACHABILITY

	A	B	C	D	E
A	0	1	1	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	0	0

NOTE: Assume A, B, C, D, E are decomposition elements. Using the connectivity matrix rules 1-6 to identify the entry *DE* as A (Rule 3) B-to-A would not be allowed (it violates Rule 2). This results in column 1 having all zeros.

A24. Test Coverage

A24.1 Application. Test coverage is a measure of the completeness of the testing process from both a developer and a user perspective. The measure relates directly to the development, integration and operational test stages of product development: unit, system and acceptance tests. The measure can be applied by developers in unit tests to obtain a measure of thoroughness of structural tests. The developer can use the program class of primitives. The system tester can apply the measure in two ways: (1) focusing on requirement primitives, the system tester can gain a user-view of the thoroughness of functional tests, or (2) the program class of primitives can be used to measure the amount of implementation in the operational environment.

A24.2 Primitives. The primitives for test coverage are in two classes, program and requirement. For program, there are two types, functional and data. The program functional primitives are either modules, segments, statements, branches (nodes), or paths. Program data primitives are equivalence classes of data. Requirement primitives are either test cases or functional capabilities.

A24.3 Implementation. Test coverage (TC) is the percentage of requirement primitives implemented times the percentage of primitives executed during a set of tests. A simple interpretation of test coverage can be expressed by the following formula:

$$TC(\%) = \frac{\text{(Implemented capabilities)}}{\text{(Required capabilities)}} \times \frac{\text{(Program primitives tested)}}{\text{(Total program primitives)}} \times 100$$

A24.4 Interpretation. Test coverage is only an indication of the thoroughness of test. Due to the interplay of data with function, there are usually an infinite number of cases to test. This can be significantly reduced, if equivalence relations can be logically defined on the data and functions.

Confidence in the level of test thoroughness is increased by using more refined primitives. For example, segment coverage of 100% is a stronger statement than 100% module coverage.

A24.5 Considerations. Test coverage is a measure of the testing made on two categories of the software product, the program and the requirement (Fig A24.5-1). The program is a set of coded

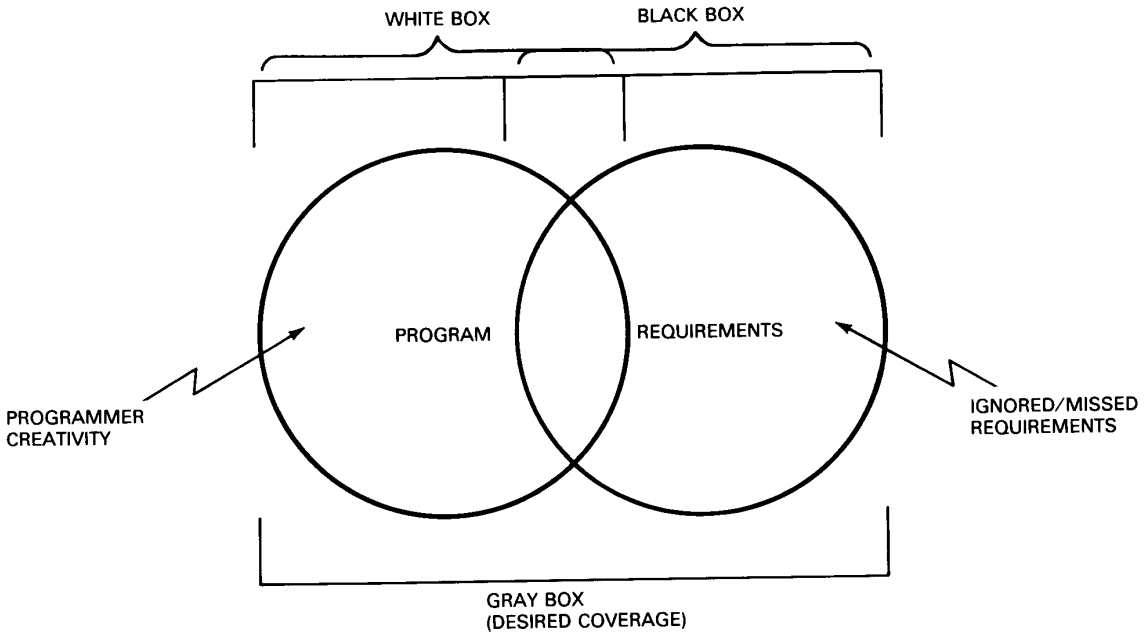


Fig A24.5-1
Test Coverage

instructions. The requirement is a set of user-intended capabilities. The intersection is the set of implemented user-intended capabilities. The left area represents either programmer creativity or those areas for which necessary implementation decisions were made due to the hardware, for example. The right area represents those user-intended capabilities that are missing in the implementation either unintentionally or intentionally due to other overriding factors such as storage. Corresponding to this model we can associate test activities and test coverage measures.

To the program, we associate a unit or functional test activity that is based on the premise that the structure of the programmed implementation is known. Hence the program primitives are known and available for a developer-oriented measure of test coverage. This is usually known as white box testing or clear box testing.

To the requirements, we associate a system test activity that in most cases assumes no knowledge of the programmed implementation. Test scenarios are designed to demonstrate correct operation from a user's view. The requirement primitives are known and can be used to give a user-oriented test coverage measure. This is usually known as black box testing.

A combination activity can be postulated. This is known as gray box testing. Test coverage in this activity is the product of the test coverage of program and requirement.

A24.6 Training. The user should understand basic testing theory.

A24.7 Example. Assuming all capabilities are implemented, Fig A24.7-1 gives the test coverage of a set of tests with segments as the primitives.

A24.8 Benefits. The test coverage measure is important in regulating effort expended to benefit achieved increasing the efficiency of the test process. Test cases can be developed to maximize the coverage measure.

A24.9 Experience. Test coverage is faciliated by code instrumented with probes to monitor usage. These, however, are programming language dependent. It is not a standard practice in program language specifications to provide this monitoring feature. Many projects are "tooling up" by customizing probes into language precompilers to automate this measure. DeMillo and Martin [A77] gives an assessment of test coverage usage in the DoD community.

No.	Module Name	Number of Setments:	(Archived) Past Tests		
			Number of Invocations	Number of Segments Hit	Percent Coverage
50:	do_one_to_one	Tio 177	12	52	29.38
51:	calculate_and_test_so	15	2	11	73.33
52:	a_to_r	7	14	4	57.14
53:	GET_CELL_FLG	3	34	2	66.67
54:	Put_txt_cell	54	20	18	33.33
55:	CUR_DOWN	5	5	3	60.00
56:	RESET_GRAPH_VARS	9	1	9	100.00
57:	set_target_loop_ctr	29	2	14	48.28
	Totals	1192	397	454	38.09

Fig A24.7-1
Example of Test Coverage Measurements*
(with Segments as Primitives)

A24.10 References. See also Ceriani, Cicu, and Maiocchi [A11].

[A77] DEMILLO, R. A. and MARTIN, R. J. *Software Test and Evaluation: Current Defense Practices Overview.* Georgia Institute of Technology, Jun 1983.

[A78] McCABE, T. *Tutorial Text: Structured Testing.* IEEE Computer Society Press, 1981.

[A79] MILLER, E. *Tutorial: Program Testing Techniques.* IEEE Computer Society Press, 1977.

A25. Data or Information Flow Complexity

A25.1 Application. This is a structural complexity or procedural complexity measure that can be used to evaluate the following:

(1) The information flow structure of large scale systems;
(2) The procedure and module information flow structure;
(3) The complexity of the interconnections between modules.

Moreover, this measure can also be used to indicate the degree of simplicity of relationships between subsystems and to correlate total observed failures and software reliability with data complexity.

A25.2 Primitives

lfi = local flows into a procedure.
datain = number of data structures from which the procedure retrieves data.
lfo = local flows from a procedure.
dataout = number of data structures that the procedure updates.
length = number of source statements in a procedure (excludes comments in a procedure.

A25.3 Implementation. Determine the flow of information between modules or subsystems or both by automated data flow techniques, HIPO charts, etc.

A local flow from module A to B exists if one of the following holds true:

(1) A calls B;
(2) B calls A and A returns a value to B that is used by B; or
(3) Both A and B are called by another module that passes a value from A to B.

Values of primitives are obtained by counting the data flow paths directed into and out of the modules.

fanin = *lfi* + datain
fanout = *lfo* + dataout.

The information flow complexity (*IFC*) is *IFC* = (fanin × fanout)^2.

Weighted IFC = length × (fanin × fanout)^2.

A25.4 Interpretation. A module with a high information flow complexity is assumed to be more likely to have faults than a module with a low complexity value.

Measurements taken during integration testing show possible areas for redesign or restructuring of excessively complex subsystems/modules, or where maintenance of a system might be difficult.

Procedure complexities show the following:

*Coverage Analyzer, Version 1.8 (80 Column).

(1) Procedures that lack functionality (may have more than one function),
(2) Stress points in the system (high information traffic), or
(3) Excessive functional complexity of the subsystem/module.

A25.5 Considerations. Evaluation of procedural complexity is better suited for structured code. This measure generally is applied first during detailed design, and re-applied during integration to confirm the initial implementation. For large-scale systems, an automated procedure for determining data flow is essential.

A25.6 Training. Familiarity with data flow and complexity analysis and some experience in software design and development is required. A background in mathematics and statistics would aid in correlation studies.

A25.7 Example

Module *A*
  Call Module *B* — 1 fanout
  Update global structure *X* — 1 fanout
  Read from file *Y* — 1 fanin
End *A*
  fanin = 1,
  fanout = 2, therefore,
  IFC = (1*2)**2 = 4

A25.8 Benefits. Designers can aim toward a lower information flow complexity for reliable systems.

The complexity can be verified early in the design and again in the testing cycle, enabling changes before extensive testing has been conducted.

High information flow complexity indicates candidate modules and procedures for extensive testing or redesign. For failure correlation studies, associate the failure with a particular procedure/module.

A25.9 Experience. Limited experience is currently available [A83]. The use of this measure is being evaluated by a NASA/Goodard software group.

A25.10 References

[A80] BIEMAN, J. W., and EDWARDS, W. R. *Measuring Software Complexity.* Technical Report 83-5-1, Computer Science Department, University of Southwestern Louisiana, Feb 1983.

[A81] BIEMAN, J. W., and EDWARDS, W. R. *Measuring Data Dependency Complexity.* Technical Report 83-5-3, Computer Science Department, University of Southwestern Louisiana, Jul 1983.

[A82] BIEMAN, J. W., BAKER, A., CLITES, P. GUSTAFSON, D., and MILTON, A. A Standard Representation of Imperative Language Program for Data Collection and Software Measures Specification. *Journal of Systems and Software*, Dec 1987.

[A83] HENRY, S., and KAFURA, D. Software Structure Metrics Base on Information Flow. *IEEE Transactions on Software Engineering*, vol SE-7 (5), Sept 1981.

A26. Reliability Growth Function

A26.1 Application. This measure provides an estimate (or prediction) of the time or stage of testing when a desired failure rate or fault density goal will be acheived. It is an indicator of the level of success of fault correction.

A26.2 Primitives

NR_k = Total number of test cases during the kth stage
NS = Total number of stages
nc_k = Total number of successful test cases during the kth stage

A26.3 Implementation. Good record keeping of failure occurrences is needed to accurately document at which stage the failure occurred. Detailed failure analysis is required to determine whether a fault was inserted by previous correction efforts. This will be used to calculate the reinsertion rate of faults (r), which is the ratio of the number of faults reinserted to the total number of faults found.

A test-sequence of software is conducted in N stages (periods between changes to the software being tested). The reliability for the kth stage is:

$$R(k) = R(u) - A/k \qquad \text{(eq A26-1)}$$

where:

$$\lim_{k \to \infty} R(k) = R(u)$$

The unknown parameter A represents the growth parameter of the model. An $A > 0$ indicates the reliability is increasing as a function of the testing stage; an $A < 0$ indicates it is decreasing. The estimation of $R(u)$ and A is done using least-square estimates. The two equations used to derive the estimates are:

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k} \right) = 0 \qquad \text{(eq A26-2)}$$

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k} \right) \frac{1}{k} = 0 \qquad \text{(eq A26-3)}$$

The simultaneous solution of these two equations provides the estimates. If failure analysis was performed to determine a fault reinsertion rate (r); a third equation can be used in conjunction with (A26-2), and (A26-3) to estimate the fault insertion rate, r.

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k(1-r)} \right) \frac{1}{k(1-r)} = 0 \qquad \text{(eq A26-4)}$$

A26.4 Interpretation. A positive number (A) or essentially monotonic increasing function indicates positive reliability growth as well as a successful fault correction program. A negative number (A) or a monotonic decreasing function indicates reliability degradation as well as an unsuccessful fault correction program.

A26.5 Considerations. Some testing and accumulation of failure experience is necessary before a first estimate of the growth function can be made. Projecting growth when moving from one testing phase to the next will generally not be applicable unless the level of testing is comparable. Those models that use the success of test cases to project growth to some reliability goal are very dependent on the randomness of testing done and the completeness of coverage.

Determination of a significant rate of growth can be a statistically difficult task. Some models circumvent this by determining the distribution function of the parameter whose growth is being estimated. This reduces the problem to hypothesis testing [A89]. However, these models are generally more difficult to implement.

A26.6 Training. Some knowledge of software development and testing is necessary if models using test cases or the reinsertion rate of faults, r, are desired. Some statistical knowledge would aid in the interpretation of results for most models. The calculation involved in some of the models will require implementation on a digital computer.

A26.7 Example. A number of practical examples can be found in LaPadula [A88].

A26.8 Benefits. Once some initial testing experience is accumulated, this metric can be used throughout the testing phase and re-estimated as needed. Some models will estimate the distribution function of the parameter whose growth is being tracked. This often simplifies hypothesis testing to determine whether a specific goal has been achieved. Additionally, in some cases the time necessary to achieve a specific reliability goal can be calculated.

A26.9 Experience. See LaPadula [A88].

A26.10 References. See also Musa, Iannino, and Okumoto [A7].

[A84] DOWNS, T. Extensions to an Approach to the Modeling of Software Testing with Some Performance Comparisons. *IEEE Transactions on Software Engineering*, vol SE-12, no 9, Sept 1986.

[A85] DOWNS, T. An Approach to the Modeling of Software Testing With Some Applications. *IEEE Transactions on Software Engineering*, vol SE-11, 1985, pp 375-386.

[A86] GOEL, A. L., and OKUMOTA, K. *Baysian Software Prediction Models.* RADCTR-78-155 (5 vols), Jul 1978.

[A87] HECHT, H. *Measurement, Estimation and Prediction of Software Reliability.* NASA-CR-145135, Aerospace Corporation, Jan 1977.

[A88] LAPADULA, L. J. *Engineering of Quality Software Systems, Volume VIII: Software Reliability Modeling and Measurement Techniques.* RADC-TR-74-325, MITRE Corporation, Jan 1975.

[A89] LARSON, H. J. *Introduction to Probability Theory and Statistical Inference.* John Wiley & Sons, Inc, 1969.

[A90] LIPOW, M. *Some Variations of a Model for Software Time-to-Failure.* TRW Systems Group, Correspondence ML-74-2260.1.9.-21, Aug 1974.

[A91] LITTLEWOOD, B., and VERRALL, J. L. A Bayesian Reliability Model with a Stochastically Monotone Failure Rate. *IEEE Transactions on Reliability*, vol R-23, no 2, Jun 1974, pp 108-114.

A27. Residual Fault Count

A27.1 Application. This measure provides an indication of software integrity.

A27.2 Primitives

t_i = observed time between the ith and $(i-1)$st failure of a given severity level

f_i = number of failures during the ith time interval of a given severity level.

A27.3 Implementation. The failure-rate class of models can be used to predict the remaining faults. This class of models is used to estimate the shape of a hypothesized hazard function from which an estimate of the number of remaining faults can be obtained [A96].

A27.4 Interpretation. Ideally, the software failure rate is constant between failures. Where inter-failure times are not independent, a non-homogeneous Poisson-type model for failure count data may be used to predict the number of remaining faults [A96]. This measure is the parameter of the Poisson distribution describing the number of failures if testing is continued.

A27.5 Considerations. Assumptions for this class of models should be verified in order to select an appropriate model to fit the data. The appropriateness of the measure depends upon the validity of the selected model. Among the most common assumptions are the following:

- Failure rate is generally decreasing.
- The numbers of failures in non-overlapping time intervals are independent and distributed exponentially with different failure rates. However, for the non-homogeneous Poisson type process, the model utilizing failure count data assumes that the number of errors has independent increments even though the times between failures are not independent.
- No two failures occur simultaneously.
- The instantaneous failure rate of software is proportional to the number of errors remaining, each of which is equally likely to cause the next failure.

A27.6 Training. The implementation of this measure requires an understanding of probability theory and numerical analysis techniques.

A27.7 Example. A numerical example using a set of failure times to compute the number of remaining faults with the Jelinski-Moranda "Deeutrophication" and non-homogeneous Poisson process type models are compared in [A96].

A27.8 Benefits. The conditional distribution of remaining faults, given a number of known faults, can be used for deciding whether the software is acceptable for release [A92].

A27.9 Experience. A case study showing an actual analysis of failure data from a Space Shuttle Software project using a non-homogeneous Poisson type process is shown in [A95]. The reference illustrates the number of failures during the mission and subsequent verification of these projections using this measure. Experience with this measure can be found also in [A97].

A27.10 References. See also Musa, Iannino, and Okumoto [A7].

[A92] GOEL, A. L., and OKUMOTO, K. Time-Dependent Error-Detection Rate Model for Software Performance Assessment with Applications. *Annual Technical Report*, Contract Number F30602-78-C-0351, Rome Air Development Center, Griffis Air Force Base, Mar 1980.

[A93] JEWELL, W. S. *Bayesian Estimation of Undetected Errors.* Oper Res Center, University of California at Berkeley, Rep ORC 83-11.

[A94] JEWELL, W. S. Bayesian Extension to a Basic Model of Software Reliability. *IEEE Transactions on Software Engineering*, vol SE-11, no 12, Dec 1985.

[A95] MISRA, P. N. Software Reliability Analysis. IBM Systems Journal, vol 22, no 3, 1983.

[A96] *Software Reliability Modelling and Estimation Technique.* RADC-TR-82-263 Final Technical Report, Oct 1982.

[A97] SRIVASTAVA, V. K., and FARR, W. The Use of Software Reliability Models in the Analysis of Operational Systems. *1986 Proceedings of the Institute of Environmental Sciences*, 32nd Annual Technical Meeting, Dallas/Fort Worth, 1986.

A28. Failure Analysis Using Elapsed Time

A28.1 Applications. This measure can be used to determine reliability growth based on the estimated number of remaining faults and software reliability. It can be used to:

- predict the total number of faults in the software
- predict the software reliability
- estimate required testing time for reliability goal
- estimate required resource allocation

A28.2 Primitives

t_i = Observed time between failures of a given severity level (eg, execution time).

A28.3 Implementation. Observed failures should be recorded and the associated faults identified. The time between failures (CPU or other unit which represents actual execution time of the software) should also be recorded.

Using the maximum likelihood estimation, the method of least squares, or the Newton-Raphson optimizing scheme, the parameters and normalizing constants of candidate models can be estimated. These can be interpreted as initial fault content (N_o), normalizing constants, and initial mean time to failure (M_o) based on the model selected. Once a model has been successfully fitted to the data using any of the well-known statistical tests (eg, goodness-of-fit), then the model can be used to estimate the remaining faults (N), MTTF (M), and reliability (R).

A28.4 Interpretation. Based on the confidence in the determination of total faults, MTTF, and reliability, decisions can be made on the need for additional testing. The testing time required to reach a specified reliability can be determined and evaluated in terms of resources required to reach the goal. Resources include testing personnel, debugging personnel and available computer time.

A28.5 Considerations. Failures must be observed and recorded to improve reliability and obtain statistically valid results. The test coverage must be random and comprehensive, and the goodness-of-fit tests should be applied to determine the validity of the selected model. Failure analysis is required to determine severity and class of failures.

A28.6 Training. A background in software testing procedures is needed to perform the necessary failure analysis. The model, including the assumptions and limitations, used to determine initial values and parameters must be understood. A program to perform parameters estimation must be developed, or obtained from the author of the model if it is available.

A28.7 Example. There are many examples available in the listed references [A99, A57].

A28.8 Benefits. This measure can be re-used throughout test and operation phases to provide continuing assessment of the quality of the software.

A28.9 Experience. Extensive experience has been accumulated on the applicaiton of the Musa Model [A102]. The performance of various models are reported in [A97].

A28.10 References. See also Musa, Iannino, and Okumoto [A7] and Farr [A59].

[A98] ANGUS, J. E. The Application of Software Reliability Models to a Major C3I System. *1984 Proceedings of the Annual Reliability and Maintainability Symposium*, Jan 1984, pp 268-274.

[A99] GOEL, A. L., and OKUMOTA, K. Time Dependent Error Detection Model for Software Reliability and Other Performance Measures. *IEEE Transactions on Reliability*, vol R-28, no 3, 1979, pp 206-211.

[A100] JELINSKI, Z. and MORANDA, P. B. Software Reliability Research. In *Statistical Computer Performance Evaluation*, W. Frieberger, ed, New York: Academic Press, 1972, pp 465-484.

[A101] MORANDA, P. B. An Error Detection Model for Application During Software Development. *IEEE Transactions on Reliability*, vol R-30, no 4, 1981, pp 309-312.

[A102] MUSA, J. D. The Measurement and Management of Software Reliability. *Proceedings of the IEEE*, vol 68, no 9, 1980, pp 1131-1143.

[A103] MUSA, J. D. and IANNINO, A. Software Reliability Modeling: Accounting for Program Size Variation Due to Integration or Design Changes. *ACM SIGMETRICS Performance Evaluation Review*, 10(2), 1981, pp 16-25.

[A104] ROSS, M. Software Reliability: The Stopping Rule. *IEEE Transactions on Software Engineering*, vol SE-11, no 12, Dec 1985, pp 1472-1476.

[A105] SUKERT, A. N. An Investigation of Software Reliability Models. *1977 Proceedings of the Annual Reliability and Maintainability Symposium*, Philidelphia, 1977.

A29. Testing Sufficiency

A29.1 Application. This measure assesses the sufficiency of software-to-software integration testing by comparing actual to predicted faults.

A29.2 Primitives

NF = Total number of faults predicted in the software
F_{it} = Total number of faults detected to date in software integration testing
F_{pit} = Number of faults detected prior to software integration testing
M_{it} = Number of modules integrated
M_{tot} = Total number of modules in final configuration

A29.3 Implementation. The fault density ($M1$) should be used to calculate the estimated total number of faults (NF) in the software. The number of faults remaining in the integrated portion of the software can then be estimated using the following equation:

$$NF_{rem} = (NF - F_{pit})\, M_{it}/M_{\text{tot}}$$

Maximum and minimum tolerance coefficients (l_1 and l_1) should be established based on user experience. Values of 0.5 and 1.5 are suggested for use as the default values.

The guidelines given in Table A29.3-1 assume that all scheduled integration tests have been run.

A29.4 Interpretation. The interpretation of testing sufficiency is given in Table A29.3-1 under "Recommended Action" and "Comments."

A29.5 Considerations. The total predicted faults (NF) should be calculated using either micro quantitative software reliability models [A108] based on path structure, or Halstead's software science measures [A106, A107] based on operators and operands. The usefulness of this measure hinges on the accuracy of the prediction of NF.

The measure assumes faults are evenly distributed between modules. Due to this assumption, caution should be exercised in applying it during early stages of integration testing (eg, less than 10% of modules integrated). Tolerance coefficients (l_1 and l_2) may be tailored to support the user's tolerance experience. Examples of experience factors are the degree of software reliability desired, and historical accuracy of the fault prediction measure used.

A29.6 Training. Some mathematical and software development background is necessary to predict total faults and to determine testing sufficiency tolerance coefficients.

A29.7 Example. Assume that 30 modules of a 100-module system are integrated for software-to-software testing. Prior to integration testing, 20 faults of a total predicted fault count of 150 were discovered. During the initial stage of integration testing, 27 more faults were discovered. The standard min/max tolerance coefficients of l_1 = 0.5 and l_2 = 1.5 were selected. Then:

$$\hat{N}F_{rem} = (150 - 20) \times (30)/100 = 39$$
$$l_1\hat{N}F_{rem} = 0.5(39) = 19.5$$
$$l_2\hat{N}F_{rem} = 1.5(39) = 58.5$$
$$F_{\text{it}} = 27$$

Therefore, the recommended action is to proceed with integration testing or the next stage of testing since $19.5 < 27 < 58.5$.

A.29.8 Benefits. The measure is relatively simple to apply, and provides an early indication of testing sufficiency.

A29.9 Experience. No published experience is available.

A29.10 References

[A106] HALSTEAD, M. H. *Elements of Software Science.* Operating and Programming Systems

Table A29.3-1
Testing Sufficiency

Measure Result	Interpretation	Recommended Action	Comments
$l_1 NS_{\text{rem}} < F_{it} < l_2 N_{\text{rem}}$	Testing sufficiency is adequate	Proceed	—
$F_{it} > l_2 NF_{\text{rem}}$	Detected more faults than expected	Proceed, but apply fault density measure for each integrated module. Re-calculate NF.	Error-ridden modules may exist that should be replaced by a redesigned module.
$F_{it} < l_1 NF_{\text{rem}}$	Detected fewer faults than expected	Apply test coverage measure (see A24) before proceeding	May not have designed adequate number or variety of tests

Series, P. J. Denning, ed, New York: Elsevier North Holland, 1977, pp 84-91.

[A107] NASA-GODDARD, *Fortran Static Source Code Analyzer Program (SAP) User Guide (Revision 1)*, Sept 1982, pp 3-31.

[A108] SHOOMAN, M. L. *Software Engineering Design/Reliability/Management.* New York: McGraw Hill, 1983, chap 5.

A30. Mean Time to Failure

A30.1 Application. This measure is used for hypothesis testing a specified MTTF requirement.

A30.2 Primitives. Mean time to failure is the basic parameter required by most software reliability modules. Computation is dependent on accurate recording of failure time (t_i), where t_i is the elapsed time between the ith and the $(i-1)$st failure. Time units used should be as precise as feasible. CPU execution time provides more resolution than wall-clock time. Thus, CPU cycles would be more appropriate for a software development environment. For an operational environment, which might require less resolution, an estimate based on wall-clock time could be used.

A30.3 Implementation. Detailed record keeping of failure occurrences that accurately track the time (calendar or execution) at which the faults manifest themselves is essential. If weighting or organizing the failures by complexity, severity or the reinsertion rate is desired, detailed failure analysis must be performed to determine the severity and complexity. Prior failure experience or model fitting analysis (eg, goodness-of-fit test) can be used to select a model representative of a failure process, and to determine a reinsertation rate of faults.

A30.4 Interpretation. This measure gives an estimate of the mean time to the next failure. A high value implies good reliability and availability of the software. This must be balanced against severity and complexity, if such weights are used.

A30.5 Considerations. The calculation of the mean of the time-to-failure distribution is dependent upon the analytical form of the failure distribution. This, in turn, is dependent upon the underlying assumptions of the failure process.

The number, description, and weights of the levels of severity and complexity should be established prior to data gathering if these factors are to be considered. Techniques for validating statements of complexity and severity should be established. Test planning must emulate operational experience.

A30.6 Training. Depending upon the number of assumptions placed on the failure process and the extent to which severity and complexity weighting is desired, the user's basic knowledge may range from little or no background in mathematics and software development to a need for extensive background in both. The simplest models are easy to implement and apply. Those that attempt to account for input space, human factors, or imperfect debugging or all three require detailed data gathering and rigorous analysis. The references (A30.10) describe various techniques, including the relevant assumptions regarding the failure processes, for some or the more sophisticated approaches.

Training requirements depend upon the type of model used. In data gathering, some background in software development would be necessary to maximize the information obtained from failure analysis. An elementary background in mathematics and reliability theory would promote better interpretation of the results, but is not essential unless assumptions require use of a complex model. The calculation for most models will require implementation on a digital computer.

A30.7 Example. A simple estimate of the MTTF is illustrated. Through prior test exerience and analysis, the user assumes a constant failure rate, with no further correction of faults to be performed. Three levels of severity failure classes are considered. Suppose the levels are labeled SF_1, SF_2, and SF_3 respectively. Every failure is then assigned to one of the classes with the respective time between failure recorded within that class. Suppose the following data is recorded:

SF_1: 180, 675, 315, 212, 278, 503, 431
SF_2: 477, 1048, 685, 396
SF_3: 894, 1422

Then a simple estimate of the mean time to failure is

$$\text{MTTF} = \sum_{k=1}^{i} \frac{t_k}{i} \quad \text{for each class,}$$

where

t_k = time between failures

and

i = number of failures in the respective severity class. The calculated estimates are:

$$MTTF_{SF_1} = \frac{2594}{7} = 370.57,$$

$$MTTF_{SF_2} = \frac{2606}{4} = 651.5,$$

$$MTTF_{SF_3} = \frac{2316}{2} = 1158$$

These values may be compared against some specified MTTF for each severity class, by any of a number of statistical techniques.

Since the data confirms the existence of a constant failure rate, the failure distribution can be assumed to be exponential; ie, $F(t)$ = 1-exp (-t/MTTF), which is the probability of failure of the software within time t.

A30.8 Benefits. This measure can be used throughout the testing and operational phase of the software, as long as data gathering efforts are maintained. A number of hardware statistical techniques are based on this measure and, if the justification for similar failure processes in the software exists, extensive modeling is available.

A30.9 Experience. Significant experience with this measure has been documented in several reports [A57, A86, A111]. The reader is cautioned to utilize the assumptions required for the various models.

A30.10 References. See also Musa, Iannino, and Okumoto [A7]; Musa [A57]; Farr [A59]; and Goel and Okumoto [A86].

[A109] CURRIT, P. A., DYER, M., and MILLS, H. D. Certifying the Reliability of Software. *IEEE Transactions on Software Reliability*, vol SE-12, no 1, Jan 1986, pp 3-11.

[A110] MUSA, J. D., and OKUMOTO, K. A. Logarithmic Poisson Execution Time Model for Software Reliability Measurement. *Proceedings of the 7th International Conference on Software Engineering*, 1984, pp 230-238.

[A111] RUSHFORTH, C., STAFFANSON, F., and CRAWFORD, A. Software Reliability Estimation Under Conditions of Incomplete Information, University of Utah, RADC-TR-230, Oct 1979.

[A112] SHOOMAN, M. L., and TRIVEDI, A. K. A Many-State Markov Model for Computer Software Performance Parameters. *IEEE Transactions on Reliability*, vol R-25, no 2, Jun 1976, pp 66-68.

[A113] SUKERT, A. N., *A Software Reliability Modeling Study*. RADC-TR-76-247, Aug 1976.

[A114] WAGONER, W. L. *The Final Report on Software Reliability Measurement Study*. Aerospace Corporation, Report Number TOR-0074 (4112), Aug 1973.

A31. Failure Rate

A31.1 Application. This measure can be used to indicate the growth in the software reliability as a function of test time.

A31.2 Primitives

t_i = Observed times between failures (eg, execution time) for a given severity level, i=1,...

f_i = number of failures of a given severity level in the ith time interval

A31.3 Implementation. The failure rate $\lambda(t)$ at any point in time can be estimated from the reliability function, $R(t)$ which in turn can be obtained from the cumulative probability distribution, $F(t)$, of the time until the next failure using any of the software reliability growth models such as the non-homogeneous Poisson process (NHPP) [A99] or a Bayesian type model [A55, A115]. The failure rate is

$$\lambda(t) = -\frac{1}{R(t)}\left[\frac{dR(t)}{dt}\right] \qquad \text{(eq A31-1)}$$

where $R(t) = 1 - F(t)$.

A31.4 Interpretation. The failure rate $\lambda(t)$ obtained from NHPP depends only upon time t and decreases continuously. However, $\lambda(t)$ derived from Littlewood's reliability growth model depends both upon time t and the value growth function, $\psi(i)$ at the ith failure. By requiring the function $\psi(i)$ to be increasing, the condition

$$\lambda(t_i) \leq \lambda(t_{i-1})$$

is met in a probabilistic sense, ie, the programmer may introduce new errors while correcting an error.

A31.5 Considerations. The underlying assumptions for the NHPP model are the following:

(1) Equal probability of exposure of each fault.
(2) The fault count type NHPP Model assumes that the number of failures detected during the non-overlapping intervals, f_i, are independent.
(3) The time between failure type NHPP model assumes that the time between the $(i\text{-}1)$st and ith failures depends upon time to the $(i\text{-}1)$st failure.

The underlying assumptions for Bayesian reliability growth model are as follows:

(1) Successive execution time between failures with different failure rates are independently exponentially distributed

ie, $pdf(t_i|\lambda_i) = \lambda_i \exp(-\lambda_i t_i)$

(2) In the Littlewood-Verrall model [A115] the successive failure rate forms a sequence of independent random variables $\{\lambda_i\}$ with gamma distribution with parameters α and $\psi(i)$

ie, $pdf(\lambda_i|\alpha,\psi(i)) =$

$$\frac{[\psi(i)]^{\alpha}\lambda_i{}^{\alpha-1}\exp(-\psi(i)\lambda_i)}{\Gamma(\alpha)}$$

(3) The failure rate is statistically decreasing.
(4) In the Littlewood stochastic reliability growth model [A55], the failure rate of the program after removal of i errors is:

$$\lambda = \lambda_1 + \lambda_2 + \ldots \lambda_{NF-i}$$

where

NF = total number of faults in the program prior to testing and $\lambda_1 \ldots \lambda_{NF-i}$ are *i.i.d.* (identically independently distributed) and each has a gamma *pdf*.

A31.6 Training. The development computer programs needed to compute this measure require an understanding of stochastic processes and familiarity with numerical analysis and optimization techniques.

A31.7 Examples. A practical application for both NHPP and Littlewood-Verrall reliability growth models for obtaining the measure of software failure rate is shown in the following examples.

In Misra [A95] a fitted mean value function is obtained for the data collected by testing space shuttle flight software consisting of over .5 million lines of source code. After 38 wk of testing (2454.7 cumulative hrs) the following estimate of the parameters of the NHPP model for the major error category were calculated:

"a" = 163.813 "b" = $0.28759 * 10^{-3}$

The failure rate decrease is described by

$$\lambda(t) = ab \exp\left\{-b\left(\sum_{i=1}^{n} t_i + t\right)\right\}$$

$$\text{for } \sum_{i=1}^{n} t_i < t \qquad \text{(eq A31-2)}$$

In [A115] the probability distribution function of the ith failure rate $\{\lambda_i\}$ is assumed to be gamma (g) with parameters α and $\psi(i)$, then

$$R(t) = \int_0^{\infty} e^{-\lambda i t} g(\lambda_i|\alpha, \psi(i)) d\lambda_i \qquad \text{(eq A31-3)}$$

and the failure rate after the ith failure is now obtained by using equation A31-1 and is given by

$$\lambda(t) = \frac{\alpha}{t + \psi(i)}, \quad t_i \le t \le t_{i+1} \qquad \text{(eq A31-4)}$$

Littlewood has examined the data set of 136 execution times reported by Musa [A57] to predict the future times to failure using a linear form of $\psi(i)$:

$$\psi(i) = \beta_1 + \beta_2 i \qquad \text{(eq A31-5)}$$

The parameters α, β_1 and β_2 obtained using maximum likelihood methods [A115] based upon the first 35 observations are:

$\alpha = 1.51890$; $\beta_1 = -0.9956$ and $\beta_2 = 7.8342$

The failure rate after the ith failure can be computed by substituting the value of the parameters α and $\psi(i)$ in equation A31-4.

$$\hat{\lambda}(t) = \frac{1.5189}{t - .9956 + 7.8342i}, \qquad \text{(eq A31-6)}$$

$$t_i \le t \le t_{i+1} (i=1, \ldots 136)$$

A31.8 Benefits. The stepwise decrease in the failure rate at each incidence of a failure model can be used to show the improvement in software reliabiilty as a result of operational usage and can also be used to estimate the test time needed to reach target reliability.

A31.9 Experience. Experience with the use of this measure is provided by Misra [A95].

A31.10 References. See also Downs [A84, A85]; Goel and Okumoto [A54]; Littlewood [A55]; Misra [A95]; and Musa [A7, A57].

[A115] LITTLEWOOD, B. Theories of Software Reliability: How Good Are They and How Can

They Be Improved? *IEEE Transactions on Software Engineering*, vol 6, no 5, 1980, pp 489-500.

[A116] MILLER, D. R. Exponential Order Statistic Models of Software Reliability Growth. *IEEE Transactions on Software Engineering*, vol SE-12, no 1, Jan 1986.

[A117] OKUMOTO, K. A Statistical Method for Quality Control. *IEEE Transactions on Software Engineering*, vol SE-11, no 12, Dec 1985, pp 1424-1430.

[A118] SINGPURWALLA, N., and SOYER, R. Assessing (Software) Reliability Growth Using a Random Coefficient Autoregressive Process and Its Ramifications. *IEEE Transactions on Software Engineering*, vol SE-11, no 12, Dec 1985, pp 1456-1463.

[A119] YAMADA, S., and OSAKI, S. Software Reliability Growth Modeling: Models and Applications. *IEEE Transactions on Software Engineering*, vol SE-11, no 12, Dec 1985, pp 1431-1437.

A32. Software Documentation and Source Listings

A32.1 Application. The primary objective of this measure is to collect information to identify the parts of the software maintenance products that may be inadequate for use in a software maintenance environment. Two questionnaires are used to examine the format and content of the documentation and source code attributes from a maintainability perspective.

A32.2 Primitives. The questionnaires examine the following primitive product characteristics:

(1) modularity
(2) descriptiveness
(3) consistency
(4) simplicity
(5) expandability
(6) testability.

Sub-characteristics include format, interface, math models, and data/control.

A32.3 Implementation. Two questionnaires; Software Documentation Questionnaire and the Software Source Listing Questionnaire, are used to evaluate the software products in a desk audit. The questionnaires are contained in *Software Maintainability—Evaluation Guide* [A120].

For the software documentation evaluation, the resource documents should include those that contain program design specifications, program testing information and procedures, program maintenance information, and guidelines used in preparation of the documentation. These documents may have a variety of physical organizations depending upon the particular source, application, and documentation requirements. The documentation will, in general, consist of all documents that reveal the software design and implementation. A single evaluation of all the documents is done. Typical questions from the software Documentation Questionnaire include the following:

- The documentation indicates that data storage locations are not used for more than one type of data structure.
- Parameter inputs and outputs for each module are explained in the documentation.
- Programming conventions for *I/O* processing have been established and followed.
- The documentation indicates that resource (storage, timing, tape drives, disks, consoles, etc.) allocation is fixed throughout program execution.
- The documentation indicates that there is a reasonable time margin for each major time-critical program function (rate group, time slice, priority level, etc.).
- The documentation indicates that the program has been designed to accommodate software test probes to aid in identifying processing performance.

For the software source listings evaluation, the program source code is evaluated. The source code may be either a high order language or assembler. Multiple evaluations using the questionnaire are conducted for the unit (module) level of the program. The units (modules) selected should represent a sample size of at least 10% of the total source code. Typical questions from the Source Listing Questionnaire include the following:

- Each function of this module is an easily recognizable block of code.
- The quantity of comments does not detract from the legibility of the source listing.
- Mathematical models as described/derived in the documentation correspond to the mathematical equations used in the modules source listing.
- Esoteric (clever) programming is avoided in this module.
- The size of any data structure that affects the processing logic of this module is parameterized.

- Intermediate results within this module can be selectively collected for display without code modification.

A32.4 Interpretation. The average scores across evaluators, test factors, product categories, and programs are used to develop a relative degree of confidence in the software documentation and the source code.

A32.5 Considerations. The software source listings and documentation evaluation assumptions are as follows:

(1) Software source listing and documentation considerations are essentially independent of application and implementation language.

(2) The evaluators must be knowledgeable in software maintenance, but do not necessarily require detailed knowledge about the specific application software.

(3) At least five independent evaluators (with software maintenance experience) must be used to provide a 75% confidence that the resulting scores indicate the actural condition of the documentation and source listing.

(4) The software source listings selected for evaluation may be from any unit level of the program that would provide conclusions that hold for the general population of the program.

A32.6 Training. Prior to an evaluation, the evaluators must participate in a calibration exercise by completing each of the questionnaires on a sample software product. This exercise is to ensure a reliable evaluation through a clear understanding of the questions and their specific response guidelines on each questionnaire. The completed questionnaires are reviewed by the evaluator group to detect and resolve areas of misunderstanding.

A32.7 Example. Table A32.7-1 demonstrates the results of an evaluation. The questionnaires employ a six-point response scale where 6 is the highest possible score (completely agree), and 1 is the lowest subjective rating (completely disagree). The bracketed score indicates that for the documentation, the instrumentation test factor (how well the program has been designed to include test aids) scored below the minimum acceptable threshold criteria.

Table A32.7-1
Example Results of a Source Listing and Documentation Evaluation

Test Factor	Documentation	Source Listings
Modularity	4.3	5.3
Descriptiveness	3.9	4.5
Consistency	4.8	5.0
Simplicity	4.7	5.3
Expandability	4.5	5.1
Instrumentation	(3.0)	4.5
Weighted Score	4.3	5.0
Combined Score	4.7	

A32.8 Benefits. The application of the questionnaires provides an identification of potential problems in the software documentation and source listings that may affect the maintainability of the software.

A32.9 Experience. The evaluation is consistently used during operational test and evaluation for most Air Force systems.

A.32.10 References

[A120] *Software Maintainability Evaluators Guide.* AFOTEC Pamphlet 800-2, vol 3, no 2, Mar 1987.[4]

[A121] PEERCY, D. A Software Maintainability Evaluation Methodology. *IEEE Transactions on Software Engineering*, no 4, Jul 1981.

A33. RELY—Required Software Reliability

A33.1 Application. At the early planning phases of a project, RELY provides a measure that makes visible the tradeoffs of cost and degree of reliability. It is recognized that reliability may vary relative to the system purpose. Man-rated software in space applications, for example, has higher reliability requirements than game software. The development processes for high reliability software, especially integration and test, should correspondingly be greater than that of average software.

A33.2 Primitives. Required reliability ratings are as follows:[5]

[4]This guide can be obtained from the Air Force Operational Test and Evaluation Center (AFOTEC/DAP), Kirtland Air Force Base, NM 87117-7001.

[5]Definition of primitives and figures from Boehm [A122], reprinted by permission.

Very low. The effect of a software failure is simply the inconvenience incumbent on the developers to fix the fault. Typical examples are a demonstration prototype of a voice typewriter or an early feasibility phase software simulation model.

Low. The effect of a software failure is a low level, easily-recoverable loss to users. Typical examples are a long-range planning model or a climate forecasting model.

Nominal. The effect of a software failure is a modereate loss to users, but a situation from which one can recover without extreme penalty. Typical examples are management information systems or inventory control systems.

High. The effect of a software failure can be a major financial loss or a massive human inconvenience. Typical examples are banking systems and electric power distribution systems.

Very high. The effect of a software failure can be the loss of human life. Examples are military command and control systems or nuclear reactor control systems.

A33.3 Implementation. Depending on the required rating, the effort required for each phase of the development cycle can be adjusted to include the processes necessary to ensure that the product achieves the reliability goal. Figure A33.3-1 shows the effort factors relative to nominal necessary to achieve each of the required reliability ratings.

This technique can be used conversely to provide a reliability rating for a product through examination of the processes used to develop it (see Table A33.3-1).

A33.4 Interpretation. Differences in required reliability effort factors are relatively small for development phases prior to, and including, code unit test. The major factor difference is in the integration and test phases. The added requirements of operational and stress testing can make a 70% difference from the nominal.

A33.5 Considerations. These effort factors were developed empirically to reflect development processes used in the 1970 timeframe. They should be updated where the data exists to reflect local development processes including the productivity benefits of tools.

A33.6 Training Required. No special training is required.

Fig A33.3-1
Effort Multipliers by Phase: Required Software Reliability

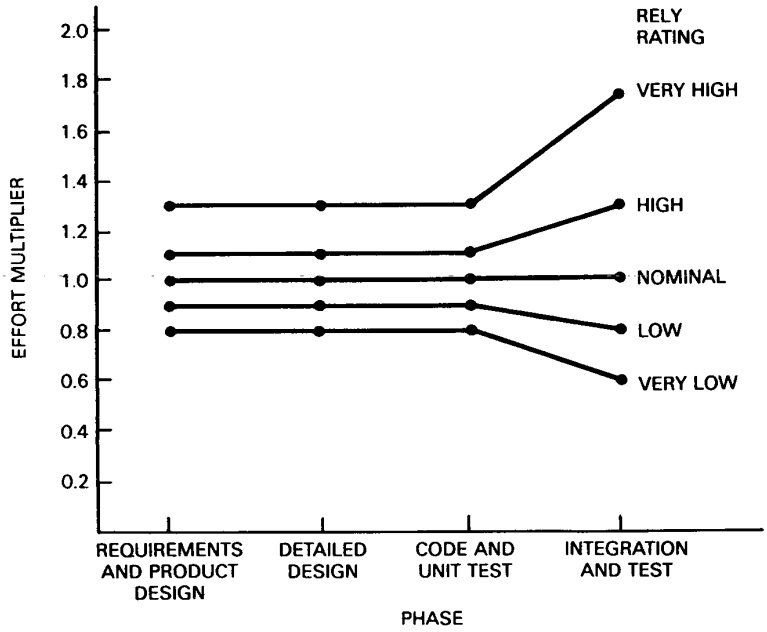


Table A33.3-1
Projected Activity Differences Due to Required Software Reliability

Rating	Requirements and Product Design	Detailed Design	Code and Unit Test	Integration and Test
Very low	Little detail Many TBDs Little verification Minimal quality assurance (QA), configuration management (CM) draft user manual, test plans Minimal program design review (PDR)	Basic design information Minimum QA, CM, draft user manual, test plans Informal design inspections	No test procedures Minimal path test, standards check Minimal QA, CM Minimal *I/O* and off-nominal tests Minimal user manual	No test procedures Many requirements untested Minimal QA, CM Minimal stress, off-nominal tests Minimal as-built documentation
Low	Basic information, verification Frequent TBDs Basic QA, CM, standards, draft user manual, test plans	Moderate detail Basic QA, CM, draft user manual, test plans	Minimal test procedures Partial path test, standards check Basic QA, CM, user manual Partial *I/O* and off-nominal tests	Minimal test procedures Frequent requirements untested Basic QA, CM, user manual Partial stress off-nominal tests
Nominal	Nominal project V & V	—	—	—
High	Detailed verification, QA, CM, standards, PDR, documentation Detailed test plans, procedures	Detailed verification, QA, CM, standards, CDR, documentation Detailed test plans, procedures	Detailed test procedures, QA, CM, documentation Extensive off-nominal tests	Detailed test procedures, QA, CM, documentation Extensive stress, off-nominal tests
Very high	Detailed verification, QA, CM, standards, PDR, documentation IV & V interface Very detailed test plans, procedures	Detailed verification, QA, CM, standards, CDR, documentation Very thorough design inspections Very detailed test plans, procedures IV & V interface	Detailed test procedures, QA, CM, documentation Very thorough code inspections Very extensive off-nominal tests IV & V interface	Very detailed test procedures, QA, CM, documentation Very extensive stress, off-nominal tests IV & V interface

A33.7 Examples. The effect of failures in an airplane control system can be the loss of human life. Hence the reliability should be specified as "Very High." To obtain this, each phase should have detailed documentation, quality assurance, adherence to standards, and verificaiton. There should be very detailed test plans and procedures, including independent verification and validation.

Suppose, however, that the Statement of Work and level of funding called for (or allowed) only basic verification, QA, draft user manuals and test plans with minimal testing. From examining Table A33.3-1, it can be seen that this corresponds to low reliability.

A33.8 Benefits. This rating is important to apply as soon as sufficient specificaiton detail is available, to justify using the intermediate level COCOMO Model for projecting costs for the software development. The ratings are consistent with a very high-level failure effects analysis and are an important input to costing and, therefore, cost-benefit tradeoffs at the earliest level.

A33.9 Experience. Experience is somewhat limited.

A33.10 References

[A122] BOEHM, B. *Software Engineering Economics*. Englewood Cliffs: Prentice-Hall, 1981, pp 374-376.

[A123] LIPOW, M. Quantitative Demonstration and Cost Consideration of a Software Fault Removal Methodology. *Journal Quality and Reliability Engineering International*, vol 1, 1985, pp 27-35.

A34. Software Release Readiness

A34.1 Application. This measure is used to quantify the user and support (consumer) risk of ownership, based on either quantitative or subjective assessments of user and support issues. Given a number of issues, one can combine them to obtain a composite risk assessment. Although the primitives described here are designed for the operation and maintenance phase, the risk assessment method is applicable during any phase of the lifecycle by the identification of appropriate issues and the selection or design of corresponding measures for assessment.

A34.2 Primitives. The measurements of effectiveness, $ME(i)$, $i=1, \ldots, I$, for issues relating to software risk assessment. Example issues are:

- Functional test coverage
- Software maturity
- Software source code listings quality factors
- Documentation quality factors
- Software operator-machine interface
- Software support resources assessments

A34.3 Implementation. This measure assumes that I issues can be combined to assess the readiness of software for release and the attendant consumer risk. For each issue i the measure $ME(i)$ is converted into an associated risk $RK(i)$ as it relates to the user and supporter. This risk is subjectively assigned by the evaluators and is defined to be in the range of 0 to 1.0, where zero denotes no risk and one denotes extreme risk. For the functional test coverage, maturity, and quality factor primitives, the associated risk is determined by:

$$RK(i) = 1.0 - ME(i)$$

Further, each of the primitives $ME(i)$ is assigned a subjective, relative weight that represents its importance to the user (supporter). For each $ME(i)$ value, there is a corresponding $RK(i)$ risk and a $W(i)$ relative importance weight. All the relative weights $W(i)$ for all user (supporter) measures sum to 1.0. Further, there is an influence matrix $[S]$ that provides a modification factor to the risk $RK(i)$ when a measure $ME(i)$ is influenced by another measure $ME(j)$. For example, consider the influence between software maturity (the fraction of software modified and added during a test [see A4.10]) and functional test coverage (the fraction of code demonstrated during a test [see A4.5]). It is clear that incomplete testing would reduce the confidence in the resulting maturity score because of incomplete demonstration of the software functions. The influence matrix $S(i,j)$ coefficients are assigned values that are representative of the dependence between $ME(i)$ and $ME(j)$.

The following steps are used in calculating this measure:

(1) Evaluate each primitive $ME(i)$.

(2) Assign a risk value $RK(i)$ to the $ME(i)$ measure such that $RK(i)$ is between zero and one (eg, $RK(i) = 1.0 - ME(i)$.)

Or, for subjective assessments, note the following:

low risk = 0.0 to 33
medium risk = .34 to .66
high risk = .67 to 1.0

(3) For each measure $ME(i)$, determine the weighting $W(i)$ that reflects the importance of the measure to the total user (supporter). A suggested default value is $1/I$ giving equal weight to all issues.

(4) For each pair of measures $ME(i)$ and $ME(j)$, assign a value for the $S(i,j)$ influence coefficient that represents the subjective or quantified influence between the measures. The $S(i,j)$ coefficients vary between zero and one, with zero representing no influence (low correlation), and one representing high dependence (high correlation). Coefficient $S(i,j)$ is the measured or subjective value of measure $ME(i)$'s dependence on measure $ME(j)$. Usually matrix [S] is symmetric.

Let the apportioned risk $RK'(i)$ for measure i in relation to the other measures be:

$$RK'(i) = \frac{\sum_{j=1}^{I} RK(j) \times S(i,j)}{\sum_{j=1}^{I} S(i,j)}$$

Then combined user (supporter) risk is:

$$R_{(\text{User/Supporter})} = \sum_{i=1}^{I} RK'(i) \times W(i)$$

A34.4 Interpretation. The value R(user) or R(supporter) is a cumulative measure of either user or supporter risk that can be compared against a predefined rating scale for reporting.

A34.5 Considerations. The quantities for $W(i)$, $S(i,j)$, and some values of $ME(i)$ are subjectively determined. It is important that the acceptance criteria be kept firmly in mind when these values

Table A34.7-1
Primitive Scores and Weighting

Measure	Measurement				Weighting $W(i)$	
	Range	Score	$ME(i)$	$RK(i)$	User W(U)	Supporter W(S)
Test coverage: $ME(1)$	0-1	0.6	0.6	0.4	0.5	0.0
Maturity index: $ME(2)$	0-1	0.83	0.83	0.17	0.5	0.0
Source listings: $ME(3)$	0-6	5.0	0.83	0.17	0.0	0.5
Documentation: $ME(4)$	0-6	4.3	0.72	0.28	0.0	0.5

Table A34.7-2
Influence Matrix (S)

User Influence Matrix — User Primitives: $ME(1)$, $ME(2)$

	$ME(1)$	$ME(2)$
$ME(1)$	1	0.6
$ME(2)$	0.6	1

Supporter Influence Matrix — Supporter Primitives: $ME(3)$, $ME(4)$

	$ME(3)$	$ME(4)$
$ME(3)$	1	1
$ME(4)$	1	1

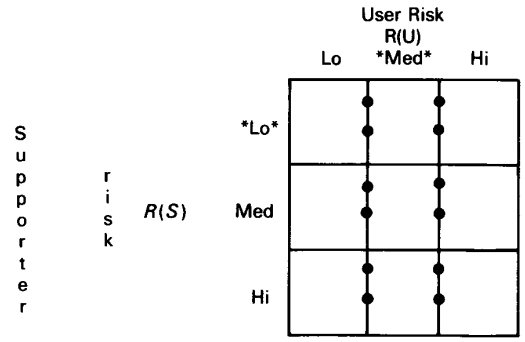


Fig A34.7-1

are established. The subjectively determined rating scale should be determined before the evaluation to avoid prejudicing the results.

A34.6 Training. Training in the evaluation/assessment of relative weights for different quality factors and for sensitivity coefficients is required, together with an understanding of the reciprocal influence of quality factors.

A34.7 Examples. Assume the primitive data values given in Table A34.7-1 are determined during testing:

Combining $ME(1)$ and $ME(2)$ for the user and $ME(3)$ and $ME(4)$ for the supporter, the influence matrices can be defined as shown in Table A34.7-2.

The above influence matrices result in the following confidence modifiers:

For the user combining $ME(1)$ and $ME(2)$,

$RK'(1) = ((0.4 * 1.0) + (0.7 * 0.6))/1.6 = 0.31$
$RK'(2) = ((0.4 * 0.6) + (0.17 * 1.0))/1.6 = 0.25$

For the supporter combining $ME(3)$ and $ME(4)$,

$RK'(3) = ((0.17 * 1.0) + (0.28 * 1.0))/2.0 = 0.23$
$RK'(4) = ((0.17 * 1.0) + (0.28 * 5.0))/2.0 = 0.23$

Calculating the user $R(U)$ and Supporter $R(S)$ risks,

$R(U) = (0.31 * 0.5) + (0.25 * 0.5) = 0.28$
$R(S) = (0.23 * 0.5) + (0.23 * 0.5) = 0.23$

The calculated $R(U)$ and $R(S)$ values represent a low-low risk respectively. The results may be presented in the combined format shown in Fig A34.7-1.

A34.8 Benefits. This approach provides a method to combine a number of measures. As such, it can be a viable approach to use in the decision process to release or buy software.

A34.9 Experience. Use of the risk assessment reporting and evaluation technique has been limited to methodology development investigations using data obtained from operational test and evaluation programs. Current efforts include a study to identify the contributing elements to the supporter risk.

A34.10 References

[A124] *Software Operator-Machine Interface Evaluators Guide.* AFOTEC Pamphlet 800-2, vol 4, Feb 1983.

[A125] *Software Support Resources Evaluation Users Guide.* AFOTEC Pamphlet 800-2, vol 5, Feb 1983.

[A126] FISK, F. B. and MURCH, W. G. *Computer Resources Risk Assessment During Operatonal Test and Evaluation.* Journal of Operational Test and Evaluation, International Test and Evaluation Association (ITEA), Jan 1985.

A35. Completeness

A35.1 Application. The purpose of this measure is to determine the completeness of the software specification during the requirements phase. Also, the values determined for the primitives associated with the completeness measure can be used to identify problem areas within the software specification.

A35.2 Primitives. The completeness measure consists of the following primitives:

B_1 = number of functions not satisfactorily defined
B_2 = number of functions
B_3 = number of data references not having an origin
B_4 = Number of data references
B_5 = number of defined functions not used
B_6 = number of defined functions
B_7 = number of referenced functions not defined
B_8 = number of referenced functions
B_9 = number of decision points not using all conditions or options or both
B_{10} = number of decision points
B_{11} = number of condition options without processing
B_{12} = number of condition options
B_{13} = number of calling routines with parameters not agreeing with defined parameters
B_{14} = number of calling routines
B_{15} = number of condition options not set
B_{16} = number of set condition options having no processing
B_{17} = number of set condition options
B_{18} = number of data references having no destination

A35.3 Implementation. The completeness measure (CM) is the weighted sum of ten derivatives expressed as:

$$CM = \sum_{i=1}^{10} w_i D_i$$

where for each $i=1,...,10$ each weight, w_i has a value between 0 and 1, the sum of the weights is equal to 1, and each D_i is a derivative with a value between 1 and 0.

To calculate the completeness measure, the definitions of the primitives for the particular application must be determined, and the priority associated with the derivatives must also be determined. This prioritization would affect the weights used to calculate the completeness measure.

Each primitive value would then be determined by the number of occurrences related to the definition of the primitive.

Each derivative is determined as follows:

$D_1 = (B_2 - B_1)/B_2$ = functions satisfactorily defined
$D_2 = (B_4 - B_3)/B_4$ = data references having an origin
$D_3 = (B_6 - B_5)/B_6$ = defined functions used
$D_4 = (B_8 - B_7)/B_8$ = referenced functions defined
$D_5 = (B_{10} - B_9)/B_{10}$ = all condition options at decision poiints
$D_6 = (B_{12} - B_{11})/B_{12}$ = all condition options with processing at decision points are used
$D_7 = (B_{14} - B_{13})/B_{14}$ = calling routine parameters agree with the called routine's defined parameters
$D_8 = (B_{12} - B_{15})/B_{12}$ = all condition options that are set
$D_9 = (B_{17} - B_{16})/B_{17}$ = processing follows set condition options
$D_{10} = (B_4 - B_{18})/B_4$ = data references have a destination

A35.4 Interpretation. The value of the completeness measure is scaled between 0 and 1 by the appropriate weights. A score near 1 is considered better than a score near 0. Those values near zero should be traced to the suspect primitive(s) to highlight any need for change in the software specification. As changes are made to the specification, the incremental specification measure values can be plotted to show if improvements are being made and how rapidly. The comparison of specification measures among different projects permits relative evaluations to be accomplished for improved completeness.

A35.5 Considerations. Each derivative helps identify characteristics in the software specification

that promote completeness during the construction of the software.

A35.6 Training. Training would be necessary to understand the definitions of the primitives (which are based on the use of a specification language, eg, PSL) for the particular application.

A35.7 Example. The following example is developed to show the numerical procedure associated with the calculation of the completeness measure.

To arrive at a value for the completeness measure, assume that the primitives have been determined to have the following values:

$B_1 = 10$	$B_7 = 8$	$B_{13} = 2$
$B_2 = 50$	$B_8 = 40$	$B_{14} = 10$
$B_3 = 50$	$B_9 = 2$	$B_{15} = 4$
$B_4 = 2500$	$B_{10} = 100$	$B_{16} = 2$
$B_5 = 2$	$B_{11} = 5$	$B_{17} = 20$
$B_6 = 50$	$B_{12} = 50$	$B_{18} = 100$

These primitives are then combined to determine the derivative values.

$D_1 = .8$	$D_5 = .98$	$D_9 = .9$
$D_2 = .98$	$D_6 = .9$	$D_{10} = .98$
$D_3 = .96$	$D_7 = .8$	
$D_4 = .8$	$D_8 = .92$	

Assuming each weight as one-tenth (1/10), the completeness measure would be calculated as follows:

$$\text{Since } CM = \sum_{i=1}^{10} w_i D_i = 0.9$$

This value could be used in comparison with other completness measures or could be recalculated following improvements made within the software specification (as additions or deletions are made to the specification).

A35.8 Benefits. The completeness measure will give an indication of the completeness (value near 1) or incompleteness (value near 0) of the software specification during the early development stages.

A35.9 Experience. Validation of the completeness measure [A50, A53] consists of determining the proper definitions and measurements of primitives and calculation of associated derivatives using PSL/PSA. Other measures of completeness are also given in Murine [A127].

A35.10 References. See also Francis [A50], and Miller et al [A53].

[A127] MURINE, G. E. On Validating Software Quality Metrics. *4th annual IEEE Conference on Software Quality*, Phoenix, Arizona, Mar 1985.

[A128] SAN ANTONIO, R, and JACKSON, K. Application of Software Metrics During Early program Phases. *Proceedings of the National Conference on Software Test and Evaluation, Feb 1982.*

A36. Test Accuracy

A36.1 Application. This measure can be used to determine the accuracy of the testing program in detecting faults. With a test coverage measure, it can be used to estimate the percentage of faults remaining.

A36.2 Primitives

N_s = Number of seeded faults

$\hat{N}_s$ = Estimated number of detectable seeded faults

A36.3 Implementation. The software to be tested is seeded with faults. The number of seeded faults detected in each testing time interval is recorded along with time to detect faults, and a mathematical model is selected to represent the cumulative seeded faults detected across the total testing period. The cumulative seeded faults detected at time infinity is then estimated and test accuracy is calculated:

$$\text{ALPHA} = \hat{N}_s / N_s$$

A36.4 Interpretation. Ideally, ALPHA is independent of test duration and approaches 100%. However, ALPHA can be used in conjunction with GAMMA, a test coverage measure [A129, A131], to estimate the percentage of inherent faults remaining at the end of test.

$$NF_{\text{rem}}(\%) = (1 - \text{ALPHA} \times \text{GAMMA}) \times 100$$

A36.5 Considerations. To meet the requirement that seeded faults reflect typical ones, a scheme for categorizing seeded faults according to difficulty of detection is required. A mathematical model must be selected that accurately describes the fault detection process. The model is likely to be complex with non-trivial estimation of its parameters, thereby requiring automation. A single model may not be appropriate for all processes.

A36.6 Training. The level of training required depends on the automated tool used. Training in mathematical model fitting and tool usage is

required. Experience in software testing would be applicable as well.

A36.7 Example. A software module to be tested was seeded with 37 faults. Given the following observed fault detection profile, a model was fitted to describe the testing process.

Model estimated seeded faults detectable at time infinity = 32; ALPHA = 32/37 = .865.

A36.8 Benefits. The test accuracy measure is an indicator of a test's ability to accurately detect faults, which will identify problem areas in testing practices.

A36.9 Experience. Limited published experience is available.

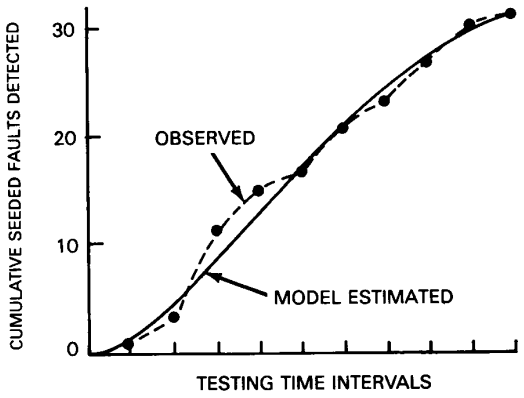
A36.10 References

[A129] OHBA, M. Software Quality = Test Accuracy × Test Coverage. *IEEE International Symposium on Software Engineering*, Chicago, 1982, pp 287-293.

[A130] OHBA, M. *Software Quality = Test Coverage × Test Accuracy.* (In Japanese.) IPS-J, Proceedings WGSE Meeting, vol 21, 1981.

[A131] SHINODA, T., ISHIDA, T., and OHBA, M. *Control Defects and Test Accuracy.* (In Japanese.) IPS-J, Proceedings Spring conference, 5C-7, 1981.

Fig A36.7-1
Control Defect Removal Process



A37. System Performance Reliability

A37.1 Application. This measure assesses the system's performance reliability, the probability that the value of each performance requirement will be less than or equal to a predefined threshold value. The specific performance requirements addressed in this measure are:

U = Utilization
X = Throughput
Q = Queue length distribution
WT = Waiting time distribution
SE = Mean server's efficiency
RT = Response time distribution
ST = Service time distribution

The application of these derived measures will ensure that reliability planning functions are done by all inter- and intra-organizations, periodic reliability etimations during the software life cycle are done to track reliability development, and verificaiton and validation are done on the system capacity.

A37.2 Primitives. The following primitives quantify the basic performance attributes that are measured over a time period of length T.

A = The number of arrivals (functional jobs) during the time period T.
SB = The total amount of time "server" is busy.
J = The number of jobs completed during T.
VR = The number of requests (visits) per functional job for each server during time period T.
T = Time period over which measurements are made.

A37.3 Implementation. These measures are applicable to operating systems and networks. Based upon the measured primitives the derived quantifiable measures are:

Γ = A/T, the average arrival rate of jobs
X = J/T, the average throughput rate
U = SB/T, the average utilization
AS = SB/J, the average service time
WT = $f(\Gamma, U, AS, VR)$, the waiting time distribution (expressed as a function of arrival rate, utilization, service time, and visit rate)
SE = $A/(AS+W)$, the server's efficiency of each job

$$R = \sum_{i=1}^{k} (VR \times S)_i + \sum_{i=1}^{k} (VR \times W)_i,$$

the total response time for each functional job over the k servers [see (5) below]

$AQ = W \times \Gamma$, the average queue length

Using any of these measures, one can derive the corresponding probability distribution of the measure under the stated assumptions of the system. The distribution can then be used to calculate the probability that the measure will achieve a desired level.

The following step by step process can be followed to calculate the probability of achieving the desired level.

(1) Determine the infrastructure of the performance model. This involves identifying the critical system resources and the system's underpinning connections in the form of a queueing network model (QNM). The basic idea of the model is that the "essential" part of the system's architecture is represented by a network of "servers" and queues that are connected by transition paths. Jobs circulate through the network, the flow being controlled by the transition probabilities.

(2) Define, qualify, and quantify the workload classes. This is to identify the resource demand characteristics of the different classes of jobs and their dynamic performance attribute matrix. This determines the apportionment of the utilization of each server by each work class.

(3) Measure the resource demands for each work class. The service time, arrival rate, and visit ratios of each system resource are measured for each class of job.

(4) Solve the QNM Model. The approximation algorithms in analytic queueing theory and operations research techniques are used to solve the model.

(5) Perform the reliability evaluation. For example, suppose the response time R must meet the goal of $R \leq R_o$ where R is given in A37.3. Calculate the system performance reliability as reliability = Probability ($R \leq R_o$) for stated assumptions during time period of length T. The system performance is not achieved if the desired reliability is not attained.

A37.4 Interpretation. In order to determine that the model did, in fact, represent the significant dynamic behavior of the system, the monitored measures from the real system need to be compared with that of the model. The validated and verified model can then be used to develop the planning strategy by answering the following questions:

(1) Where are the bottlenecks in the system and why do they exist?

(2) What actions are required to rectify the deficiencies?

(3) What performance improvement can be expected by upgrading the system level software efficiency for a given scheduling policy?

(4) How will the addition of a new application functionality affect the current performance?

(5) What are the single failure paths in the system configuration and their effects on downtime? The system downtime is qualified and quantified as the elapsed time during which a degradation or a cessation of system performance occurs.

A37.5 Considerations. There are two basic operational techniques to the deployment of the monitors for the primitives.

(1) The event trace mode records all state transition events that can be used for a complete "reconstruction" of the critical activities via a trace driven simulation program.

(2) The sampling mode involves the gathering and sampling states of the system at predetermined times. A "steady state" condition must be assumed. The performance model must be validated via measurements before the model can be used with a high degree of confidence.

A37.6 Training. Knowledge of probability, statistics, queuing theory, hardware and operating system software architecture are required.

A37.7 Example. This example is a simple one-server queueing system taken from Denning and Buzen [A137]. This simple system could represent some type of computer device (ie, disk unit, I/O device, etc.) within a more complex computer design. See Denning and Buzen [A137] for more complex examples using these measures for performance and reliability analysis.

Fig A37.7-1 is a pictorial representation of a simple computer system in which requests are being made to a device. The device processes

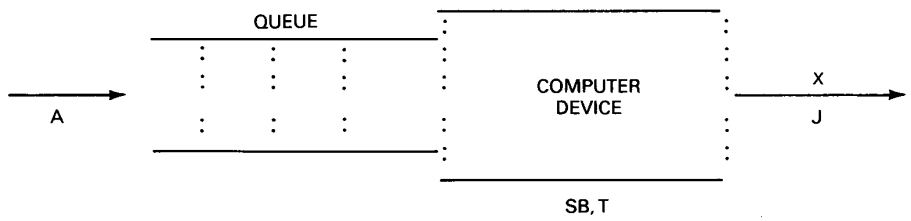


Fig A37.7-1
Single Server System

those requests immediately if it is free; otherwise it files the request in a queue to be processed when it becomes free.

Suppose the device is observed over a period of $T = 20$ sec. When the device is first observed there are two requests in the queue and a third has just entered processing. During this time period seven new requests are made of the device (ie, A = 7). It is also observed that the device was busy processing the requests for a period of 16 seconds and that during this time period all ten requests were completed. Hence B = 16 and C = 10.

The basic performance measures can then be calculated as:

$\Gamma = A/T = 7/20 = .35$ jobs per second

$X = J/T = 10/20 = .5$ jobs per second for throughput

$U = SB/T = 16/20 = .8$ utilization factor

$AS = SB/J = 16/10 = 1.6$ seconds for the average service time

See Denning and Buzen [A137] for the derivation of the remaining measures.

A37.8 Benefits. These measures can be used throughout the system life cycle to ensure an effective and resource-efficient design. These measures will provide the logical and economic framework for critical tradeoff decisions, (eg, performance vs functionality, cost vs performance, etc).

A37.9 Experience. Many examples can be found in the literature on queueing theory of this network analysis approach [A133, A135, A136, and A137].

A37.10 References

[A132] ALLEN, A. O. *Probability, Statistics and Queueing Theory.* Academic Press, 1978.

[A133] ALLEN, A. O. Queueing Models of Computer Systems. *Computer*, vol 13, no 4, Apr 1980.

[A134] BARD, Y. An Analytic Model of the VM/370 System. *IBM Journal of Research and Development*, vol 22, no 5, Sept 1978.

[A135] BASKETT, F., CHANDY, K. M., and PALACIOS, F. G. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Communications of the ACM*, vol 22, no 2, Apr 1975.

[A136] BUZEN, J. P., and DENNING, P. J. Measuring and Calculating Queue Length Distributions. *Computer*, vol 13, no 4, Apr 1980.

[A137] DENNING, P. J., and BUZEN, J. P., The Operational Analysis of Queueing Network Models. *Computing Surveys*, vol 10, no 3, Sept 1978.

[A138] GRAHAM, G. S. Queueing Network Models of Computer System Performance. *Computing Surveys*, vol 10, no 3, Sept 1978.

[A139] TRIVEDI, K. S. and KINICKI, R. E. A Model for Computer Configuration Design. *Computer*, vol 14, no 4, Apr 1980.

A38. Independent Process Reliability

A38.1. Application. This measure provides a direct method to compute reliability (R) of certain types of software systems such as telephone switching and business transaction systems in which the programs have no loops and no local dependencies exist. This is a measure of the reliability of service that a software system provides.

A38.2 Primitives

f_{ij} = frequency of execution of intermodule transfer from module i to j.

MD = number of modules

P_i = ith process which can be generated in a user environment.

q_i = probability that P_i will be generated in a user environment.

r_i = random variable equal to 1 if the process P_i generates the correct software system output and zero otherwise.

R_i = reliability of the ith module.

A38.3 Implementation. This measure assumes that a large program is composed of logically independent modules which can be designed, implemented and tested independently.

Let L = set of processes P_i that can be generated by the software system corresponding to different input values. Then the reliability R of the software system can be computed from

$$R = \sum_{\forall P_i \in L} q_i r_i$$

In large programs, it is infeasible to evaluate r_i and q_i for each process P_i because L could be large. To circumvent this difficulty, a simple Markov model is developed with intermodule probabilities, (f_{ij}), as the user profiles.

The reliability parameters of the program are expressed in terms of transitional probability parameters between states. The reliability R of the software system is defined as the probability of reaching the correct state [after module n] from the initial state [from module 1] of the Markov process, ie:

$$R = S(1,MD)\, R_n$$

where $S = [I - Q]^{-1}$;

Q is the state transition matrix spanning the entire Markov graph program; I is the identity matrix.

Elements of the matrix, $f_{ij}R_i$ represent the probability that execution of module i produces the correct result and transfers control to module j. R_i is determined from experimentation as the ratio of the number of output sets generated by module i to the number of input sets when the software system is tested by samples of representative valid input.

A38.4 Interpretation. The reliability measure provides the user with a direct estimate of the confidence in system software performance and indicates whether further testing and software updates are required for a given operational environment. In addition, this approach also indicates to the user how to improve the system reliability most effectively by evaluating the sensitivity of the system reliability with respect to that of a module [A142]. A high-sensitivity coefficient indicates a critical module that has greater impact on software reliability.

A38.5 Considerations. The underlying assumptions for the use of this measure are the following:

(1) The reliability of the modules are independent — ie, one module fault is not compensated by another.

(2) The transfer of control between program modules is a Markov process. (Next module to be executed depends probabilistically on the present module only and is independent of past history). When no modification is made on the modules, the transition probabilities are assumed constant. These characterize the user environment with respect to the reliability model.

A38.6 Training. The implementation of this methodology requires an understanding of stochastic processes.

A38.7 Examples. An example of the use of this measure is illustrated in Cheung [A142].

A38.8 Benefits. The measure gives a direct indication of the reliability of service to the user and determines whether further testing and improvement is required for the application. The sensitivity coefficient provides a scheme for correcting faults by providing priority updates to modules that have the greatest impact on software reliability.

A38.9 Experience. Limited published experience is available.

A38.10 References

[A140] CHEUNG, R. C. A Structural Theory for Improving SW Reliability. Ph.D. Dissertation, University of California at Berkeley, Dec 1974.

[A141] CHEUNG, R. C. and RAMAMOORTHY, C. V. Optimal Measurement of Program Path Frequencies and Its Applications. In Proc. 1975, *Int Fed Automat Contr Congr*, Aug 1975.

[A142] CHEUNG, R. C. A User Oriented Software Reliability Model. *IEEE Transactions on Software Engineering*, vol SE-6, no 2, Mar 1980.

A39. Combined Hardware and Software (System) Operational Availability

A39.1 Application. This measure provides a direct indication of system dependability, that is, whether it is operating or operable when required by the user.

A39.2 Primitives

λ = Observed software failure rate

β = Observed hardware failure rate

μ_i = Observed software fault correction rate with i faults in the software

γ = Observed hardware repair rate

NF = Estimated number of remaining software faults

P_s = Probability of correcting a software fault when detected ($0 \leq P_s \leq 1$)

P_h = Probability of correctly repairing a hardware failure ($0 \leq P_h \leq 1$)

A39.3 Implementation. The basic Markovian model is used for predicting combined hardware and software (system) availability as a function of operating time. System upstate probabilities $P_{NF,n}(t)$ with n remaining faults, given that there were NF software faults at the commissioning (t=0) of the system, is computed for n = 0, 1, 2, ..., N.

$$\text{System availability } A(t) = \sum_{n=0}^{NF} P_{N,n}(t)$$

The expression for $P_{NF,n}(t)$, n=1, ..., NF is derived as follows [A96]:

$$P_{NF,n}(t) = G_{NF,n}(t) - \sum_{j=1}^{K} \frac{\prod_{i=n+1}^{NK} (P_s\lambda_i\mu_i)(-x_j+P_h\gamma)^{NF-n} A(-x_j)(1-e^{-x_jt})}{x_j \prod_{\substack{i=1 \\ i\neq j}}^{K} (-x_j+x_i)}$$

where

$G_{NF,n}(t)$ = Cumulative distribution function of the first passage time for operational state with NF faults to operational state with n faults and is expressed as:

$$\sum_{j=1}^{K} \frac{\prod_{i=n+1}^{NF} (P_s\lambda_i\mu_i)(-x_j+P_h\gamma)^{NF-n}}{\prod_{\substack{i=1 \\ i\neq j}}^{K} (-x_j+x_i)} \frac{1}{-x_j}(e^{-x_jt}-1)$$

$\lambda_i = i\lambda$

$\mu_i = i\mu$

$K = 3\,(NF-n)$

$x_1 = x_{1,(n+1)}$

$x_2 = x_{2,(n+1)}$

$x_3 = x_{3,(n+1)}$

$x_4 = x_{1,(n+2)}$

$x_5 = x_{2,(n+2)}$

$x_6 = x_{3,(n+2)}$

.

.

.

$x_{1(NF-n)} = x_{1,NF}$

$x_{2(NF-n)} = x_{2,NF}$

$x_{3(NF-n)} = x_{3,NF}$

$-x_{1,i}$, $-x_{2,i}$ and $-x_{3,i}$ (i=n+1, ..., NF) are the roots of the polynomial

$$S^3 + S^2(\lambda + \mu + \beta + P\gamma) + S(P\lambda\mu + \beta\mu + \lambda P\gamma) + \mu P_h\gamma) + P_sP_h\gamma\lambda_i\mu_i$$

Also,

$$A(-x_j) = (-x_j\beta)(-x_j+\mu_n) + \lambda_n(-x_j+P_s\mu_n)(-x_j+P_h\gamma)$$

A39.4 Interpretation. $A(t)$ will show a characteristic drop initially and then will rise, approaching a steady state value determined by the hardware availability.

A39.5 Considerations. The underlying assumptions for the use of this measure based on *Software Reliability Modelling and Estimation Technique* [A96] are as follows:

(1) The occurrence of software and hardware failures are independent.

(2) The probability of two or more software or hardware failures occurring simultaneously is negligible.

(3) Failures and repairs of the hardware are independent of both fault detection and correction of the software.

(4) The software failure rate and the hardware failure rate are constants.

(5) The time to repair hardware failures and the time to correct software failures follow exponential distributions, each with a fixed parameter.

A39.6 Training. The implementation of this methodology requires an understanding of stochastic processes and familiarity with programming analytical models.

A39.7 Example. See combined HW/SW Reliability Models, Section A2.4.2 [A143] for a specific example for system availability.

A39.8 Benefits. A sampling measure for the average availabiity $A_{av}(t)$ of an operational system can be developed and compared with the theoretically computed figure for availability $A(t)$.

A39.9 Experience. Few hardware/software models have been developed that can be applied to the measurement of system operational availability (see Software Reliability Modelling and Estimation Technique [A96]). These models have not yet been tested against actual system failure data. Yet, because it is judged to have great potential for addressing the problem of measuring combined hardware/software performance, their application to actual system failure data is encouraged. However, as other models become available, they too should be considered when system availability calculations/predictions need to be made [A144].

A39.10 References. See also *Software Reliability Modelling and Estimation Technique* [A96].

[A143] *Combined HW/SW Reliability Models*, RADC-TR-82-68, Final Technical Report, Apr 1982.

[A144] SUMITA, U., and MASUDA, Y. Analysis of Software Availability/Reliability Under the Influence of Hardware Failures. *IEEE Transactions on Software Reliability*, vol SE-12, no 1, Jan 1986.