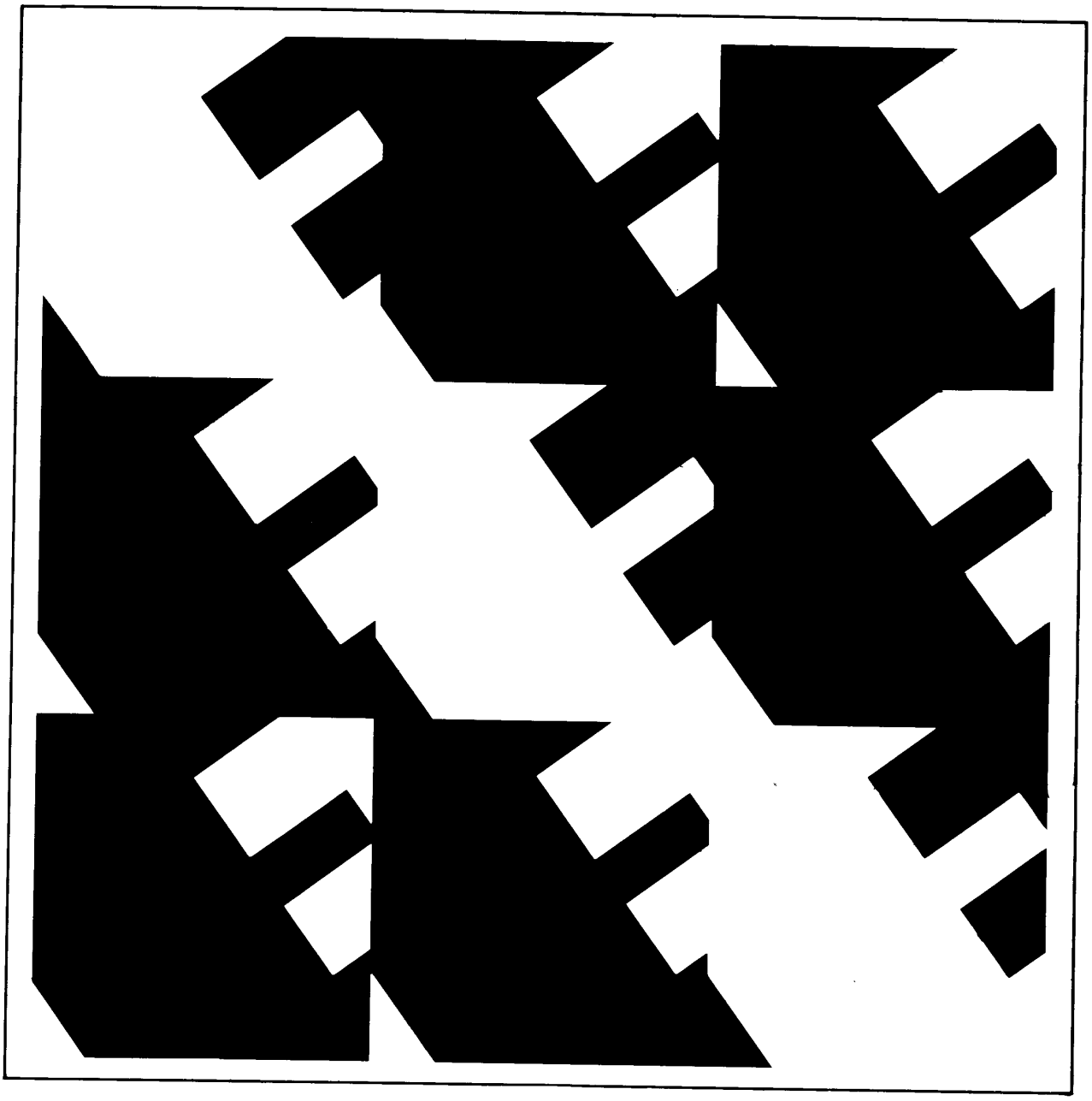
IEEE Standard Dictionary of Measures to Produce Reliable Software



IEEE Std 982.1-1988

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

April 30, 1989

SH12542

IEEE Standard Dictionary of Measures to Produce Reliable Software

Sponsor

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

Approved June 9, 1988

IEEE Standards Board

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017-2394, USA

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 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.)

This standard provides a set of measures indicative of software reliability that can be applied to the software product as well as to the development and support processes. It was motivated by the need of software developers and users who are confronted with a plethora of models, techniques, and measures. There is a need for measures that can be applied early in the development process that may be indicators of the reliability of the delivered product. This standard provides a common, consistent definition of a set of measures that may meet those needs.

This standard is intended to serve as the foundation on which researchers and practitioners can build consistent methods. This document is designed to assist management in directing product development and support toward specific reliability goals. Its 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 specific environments.

There is both a process goal and a product goal for this document:

(1) The process goal is to provide measures that may be applicable throughout the life cycle and may provide the means for continual self-assessment and reliability improvement.

(2) The product goal is to increase the reliability of the software in its actual use environment during the operations and support phases.

This standard is intended to be of interest to design, development, evaluation (for example, auditors, procuring agency) and maintenance personnel; software quality and software reliability personnel; and to operations and acquisition support managers. This document contains four sections:

Section 1: Scope, establishes the goals and boundaries of the standard.

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

Section 3: Functional Classification of Measures, provides a taxonomy with respect to measure objectives.

Section 4: Measures for Reliable Software, presents the measures ordered in general by complexity.

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

James Dobbins, *Chairman*

Ray Leber, *Co-Chairman*

Ted Workman, *Co-Chairman*

Peter Bright
Antonio Cicu
Carlo Cucciati
Walter Ellis
William Farr
Stuart Glickman
Laura Good
Nancy Hall
Philip Marriott
Roger Martin
Walter Murch
Patricia Powell
Frank Salvia
David Siefert
Raghu Singh
Vijaya Srivastava
Henry Trochesset
William Wilson

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 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

When the IEEE Standards Board approved this standard on June 9, 1988, it had the following membership:

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

Andrew G. Salem, *Secretary*

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

*Member Emeritus

Contents

FIGURES

TABLES

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 (modular) test requirements	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
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 Standard Dictionary of Measures to Produce Reliable Software

1. Introduction

1.1 Scope. The objective of this standard is to provide the software community with defined measures currently used as indicators of reliability. The predictive nature of these measures is dependent on correct application of a valid model.

This standard presents a selection of applicable measures, the proper conditions for using each measure, the methods of computation, and the framework for a common language among users. It moreover provides the means for continual assessment of the process, as well as the products, at each phase of the life cycle. By emphasizing early reliability assessment, this standard supports methods through measurement to improve the product reliability. Successful application of the measures is dependent upon their use as intended in the specified environments.

Applicability is not restricted by the size, type, complexity, or criticality of the software. Typical life cycle phases based upon ANSI/IEEE Std 729-1983 [1],[1] are used to illustrate when each of the measures may be applied.

Errors, faults, and failures serve as primitive units for the majority of the measures.

1.2 References. The following publications shall be used in conjunction with this standard:

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

[2] Software Maintainability — Evaluation Guide, AFOTEC Pamphlet 800-2, vol 3, Mar 1987.[3]

2. Definitions

Most of the definitions listed below can be found in ANSI/IEEE Std 729-1983 [1], IEEE Standard Glossary of Software Engineering Terminology. The definitions of the terms *concept phase*, *defect*, *measure*, *primitive*, and *software management* are not present in ANSI/IEEE Std 729-1983 [1], and establish meaning in the context of this standard. 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, incorrect translation, or omission of a requirement in the design specification (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]).

[1] The numbers in brackets correspond to those of the references listed in 1.2.

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

[3] AFOTEC publications are available from the Air Force Operational Test and Evaluation Center (AFOTEC/DAP), Kirtland Air Force Base, NM 87117-7001.

(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 (ANSI/IEEE Std 729-1983 [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 encounterred, may cause a failure. Synonymous with **bug** (ANSI/IEEE Std 729-1983 [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. 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, and 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 (ANSI/IEEE Std 729-1983 [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 ot maximize reliability in light of project constraints such as resources (cost), schedule, and performance.

3. Functional Classification of Measures

The measures can be divided into two functional categories: product and process (Table 3-1). Product measures are applied to the software objects produced and are divided into six subcategories. Process measures are applied to the activities of development, test, and maintenance. Process measures are divided into three subcategories.

3.1 Product Measures. The product measures address cause and effect of the static and dynamic aspects of both projected reliabilty 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. These product measures cover more than the correctness aspect of reliability; they also address the system utility aspect of reliability. The following six product measure subcategories address these dimensions of reliability:

(1) Errors; Faults; Failures — Count of defects with respect to human cause, program bugs, 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 Product 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.

3.2 Process Measures. The three process measure subcategories are directly related to process management:

(1) Management Control — The assessment of guidance of the development and maintenance processes.
(2) Coverage — The assessment of the presence of all necessary activities to develop or maintain the software product.
(3) Risk; Benefit; Cost Evaluation — The assessment of the process tradeoffs of cost, schedule, and performance.

The measure classification matrix (Table 3-1) is a cross index of the measures and functional categories and can be used to select measures applicable to each category.

4. Measures for Reliable Software

4.1 Fault Density

4.1.1 Application. This measure can be used to

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

Table 3-1
Measure Classification Matrix

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

(2) Determine if sufficient testing has been completed based on predetermined goals for severity class.
(3) Establish standard fault densities for comparison and prediction.

4.1.2 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 nonexecutable data declarations in thousands

4.1.3 Implementation. Establish severity, failure types, and fault types.

(1) Failure types might include *I/O* 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 nonexecutable data declaration source code (KSLOC).
(5) Calculate the fault density for a given severity level as

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

4.2 Defect Density

4.2.1 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.

4.2.2 Primitives. Establish severity levels for defect designation.

D_i = total number of unique defects detected during the ith design or code inspection process
I = total number of inspections
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 nonexecutable data declarations in thousands

4.2.3 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.

4.3 Cumulative Failure Profile

4.3.1 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 acceptability reliable system.
(3) Identify modules and subsystems that require additional testing.

4.3.2 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, ...

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

4.4 Fault-Days Number

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

4.4.2 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, the time unit can be made relative to test time or operational time.

4.4.3 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 num-

ber 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 4.4.3-1 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 knowledge of when the corresponding piece of requirement was specified, is not known.

The measure is calculated as shown in Fig 4.4.3-1.

4.5 Functional or Modular Test Coverage

4.5.1 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 the 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.

4.5.2 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

4.5.3 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 the users (developers) requirements. This ratio is expressed as

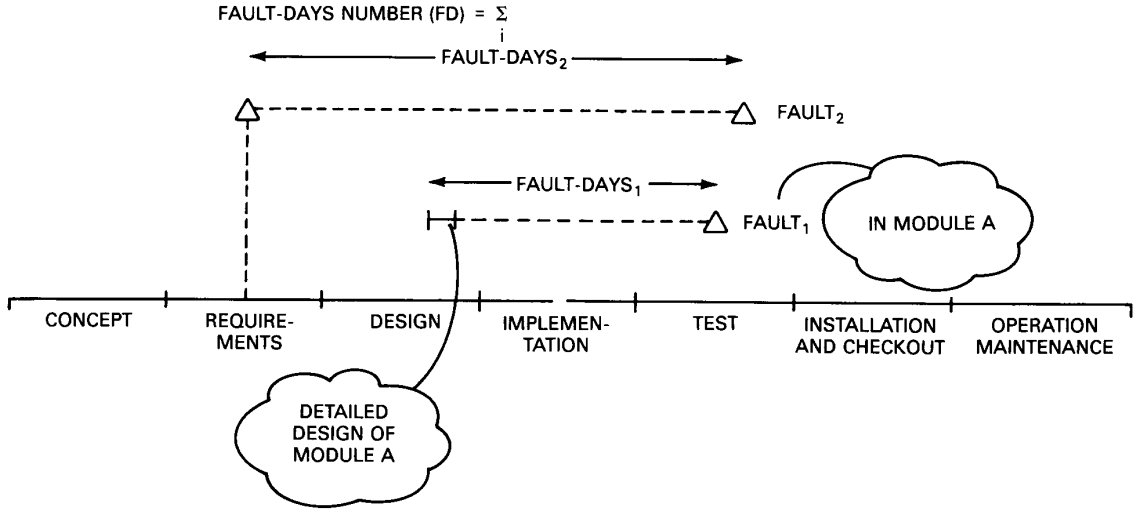
$$\text{FUNCTIONAL (MODULAR) TEST COVERAGE INDEX} = \frac{FE}{FT}$$

4.6 Cause and Effect Graphing

4.6.1 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 time of computing application where the specification is clearly stated (that is, no ambiguities) and combinations of input conditions can be identified. It is used in developing and designing test cases that have a

Fig 4.4.3-1
Calculation of Fault-Days



high probability of detecting faults that exist in programs. It is not concerned with the internal structure or behavior of the program.

4.6.2 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

4.6.3 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:

(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 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.

4.7 Requirements Traceability

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

4.7.2 Primitives

$R1$ = number of requirements met by the architecture

$R2$ = number of original requirements

4.7.3 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):

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

4.8 Defect Indices

4.8.1 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.

4.8.2 Primitives. For 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)

4.8.3 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:

$$DI = \sum_i (i * PI_i)/PS$$

$$= (PI_1 + 2PI_2 + 2PI_3 + \ldots + iPI_i + \ldots)/PS$$

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 (that is, 2 or 3, respectively) assigned.

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

4.9 Error Distribution(s)

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

4.9.2 Primitives. Error description:

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

4.9.3 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 4.9.3-1. In the three examples of Fig 4.9.3-1, the errors are classified and counted by phase, by the cause, and by the cause for deferred fault detection. Other similar classification 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.

4.10 Software Maturity Index

4.10.1 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.

4.10.2 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

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

4.10.3.1 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 baseline (M_T).

Calculate the maturity index:

$$\text{MATURITY INDEX} = \frac{\text{NUMBER OF CURRENT FUNCTIONS (MODULES)} - \left(\begin{array}{c}\text{NUMBER OF CURRENT BASELINE FUNCTIONS (MODULES) THAT HAVE BEEN ADDED} \\ + \text{NUMBER OF FUNCTIONS (MODULES) CHANGED} \\ + \text{NUMBER OF CURRENT BASELINE FUNCTIONS (MODULES) THAT HAVE BEEN DELETED}\end{array}\right)}{\text{NUMBER OF CURRENT FUNCTIONS (MODULES)}}$$

that is

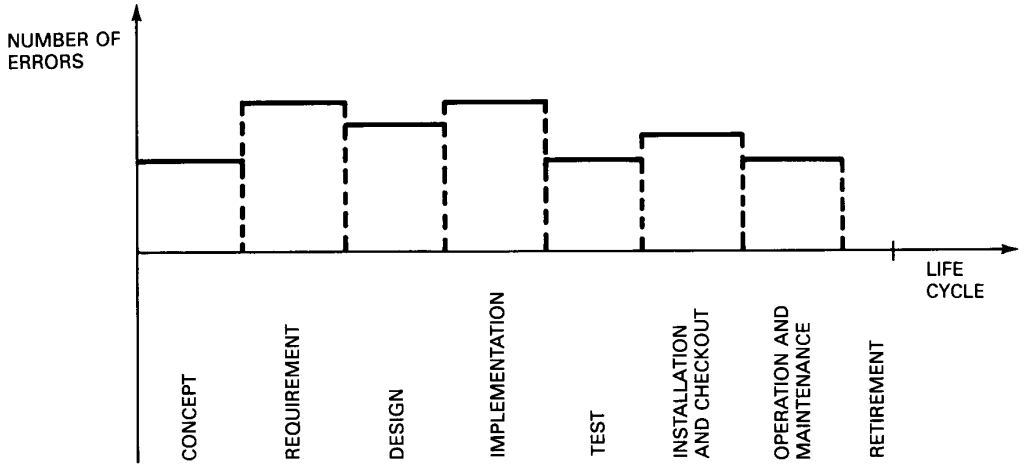
$$SMI = \frac{M_T - (F_a + F_c + F_{del})}{M_T}$$

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).

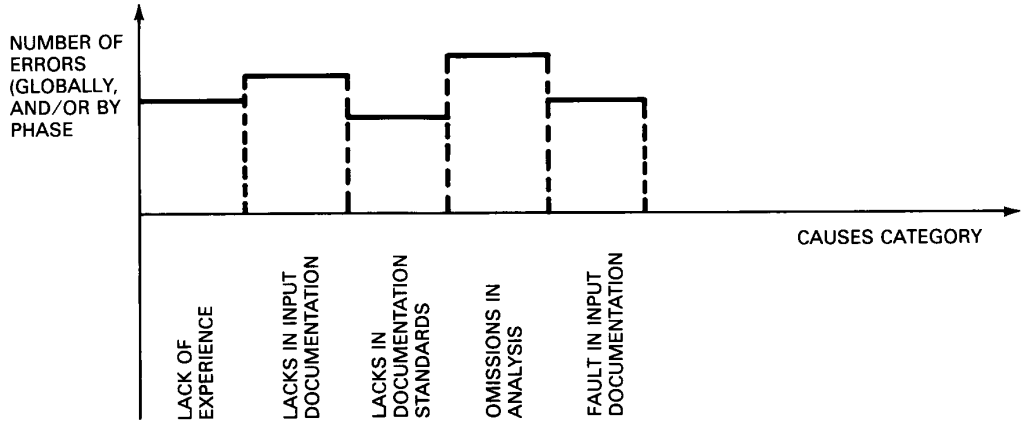
4.10.3.2 Implementation #2. The software maturity index may be estimated as

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

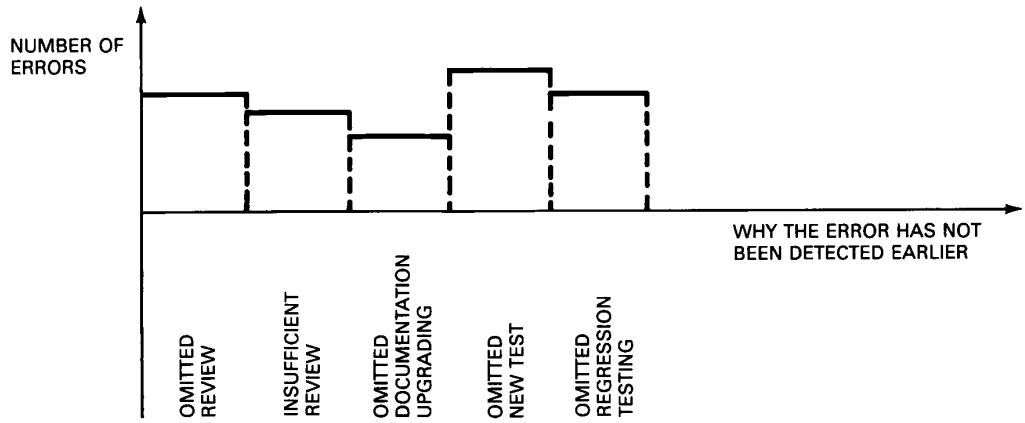
The change and addition of functions (modules) is tracked and the maturity index is calculated



(a) Error Distribution by Phase



(b) Error by Cause Category



(c) Suggested Causes for Error Detection Deferral

Fig 4.9.3-1
Error Analysis

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

4.11 Manhours per Major Defect Detected

4.11.1 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.

4.11.2 Primitives

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

T_2 = time expended by the inspection team in conduct of a design or code inspection meeting

S_i = number of major (nontrivial) defects detected during the ith inspection

I = total number of inspections to date

4.11.3 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 categores. (A major defect is one which 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 8000 lines of detailed design or code have been inspected.

4.12 Number of Conflicting Requirements

4.12.1 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.

4.12.2 Primitives

(1) List of the inputs

(2) List of the outputs

(3) List of the functions performed by the program

4.12.3 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.

4.13 Number of Entries and Exits per Module

4.13.1 Application. This measure is used to determine the difficulty of a software architecture, as represented by entry and exist points identified in a modular specification or design language.

4.13.2 Primitives

e_i = number of entry points for the ith module

x_i = number of exit points for the ith module

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

$$m_i = e_i + x_i$$

4.14 Software Science Measures

4.14.1 Application. These measures apply the 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.

4.14.2 Primitives

$n1$ = number of distinct operators in a program

$n2$ = number of distinct operands in a program

$N1$ = total number of occurrences of the operators in a program

$N2$ = total number of occurrences of the operands in a program

4.14.3 Implementation. Once the source code has been written, this measure can be applied to predict the program difficulty and other derived quantities using the following equations:

Program vocabulary: $l = n1 + n2$

Observed program length: $L = N1 + N2$

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

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

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 18 elementary mental discriminations per second)

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

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

4.15 Graph-Theoretic Complexity for Architecture

4.15.1 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:

(1) Static Complexity — A measure of software architecture, as represented by a network of modules, useful for design trade-off analyses. Network complexity is a function based on the countable properties of the modules (nodes) and intermodule connections (edges) in the network.

(2) Generalized Static Complexity — 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.

(3) Dynamic Complexity — 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.

4.15.2 Primitives

K = number of resources, index by $K = 1, \ldots, K$

E = number of edges, indexed by $i = 1, \ldots, E$

N = number of nodes, indexed by $j = 1, \ldots, N$

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 $R(K,E)$

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

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

4.15.3 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.

(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

$$C = E - N + 1$$

(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

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

(3) Dynamic Complexity — A change in the number of edges may result from module interruption due to invocations and re-

turns. 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 (ex. time average) is then used to indicate the evolution of the complexity of the software.

4.16 Cyclomatic Complexity

4.16.1 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.

4.16.2 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)

4.16.3 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.

4.17 Minimal Unit Test Case Determination

4.17.1 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.

4.17.2 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 1 edge emanating from it

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

4.17.3 Implementation. The cyclomatic complexity is first computed using the cyclomatic complexity measure described in 4.16. 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.

4.18 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.

4.18.1 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.

4.18.2 Primitives

NR = number of runs made 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)), which constitutes a run)

P = probability measure over the sample space

P_i = probability that the ith run is selected from the sample space

4.18.3 Implementation

4.18.3.1 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.

4.18.3.2 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.

4.18.3.3 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.

4.18.3.4 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 $P_r = nc/NR$.

4.18.3.5 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 $R_k = P_r^k$.

4.19 Design Structure

4.19.1 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.

4.19.2 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 nonunique database elements
$P6$ = number of database segments (partition of the state)
$P7$ = number of modules not single entrance/single exit

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

$D1$ = design organized top down (Boolean)
$D2$ = module dependence ($P2/P1$)
$D3$ = module dependent on prior processing ($P3/P1$)
$D4$ = database size ($P5/P4$)
$D5$ = database compartmentalization ($P6/P4$)
$D6$ = 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 ($\Sigma W_i = 1.0$).

4.20 Mean Time to Discover the Next K Faults

4.20.1 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.

4.20.2 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, $i=1, \ldots$

4.20.3 Implementation. The mean-time-to-failure (see 4.30) should be used to estimate the mean time to discover the next K faults in the software. This estimate can then be expressed as

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

where 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. Suppose that estimate is

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

Notice that if $K=1$, then we simply have the estimate of the MTTF (see 4.30) between the fth and $(f+1)$st failure. Then the estimate is

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

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.

4.21 Software Purity Level

4.21.1 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.

4.21.2 Primitives

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

f = total number of failures in a given time interval

4.21.3 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 failure 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.

4.22 Estimated Number of Faults Remaining (by Seeding)

4.22.1 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.

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 nonseeded faults are found.

4.22.2 Primitives

N_s = number of seeded faults

n_s = number of seeded faults found

n_F = number of faults found that were not intentionally seeded

4.22.3 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 occurrences expected under a particular set of software development conditions. 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 report fault should be reviewed to determine if it is in the class of faults being studied and, if so, if it is a seeded or an indigenous fault. The maximum likelihood estimate of the number of indigenous (unseeded) faults in the specified class is

$$\hat{NF} = \frac{n_F N_s}{n_s}$$

where $\hat{NF}$ is truncated to the integer value. The estimate of the remaining number of faults is then

$$\hat{NF}_{\text{rem}} = \hat{NF} - n_F$$

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 at "y" at a time. Using this relation one can calculate confidence intervals.

4.23 Requirements Compliance

4.23.1 Application. This analysis is used to verify requirements compliance by using system verification diagrams (*SVD*s), a logical interconnection of stimulus response elements (for example, stimulus and response), which detect inconsistencies, incompleteness, and misinterpretations.

4.23.2 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 *SVD*s:

N_1 = number due to inconsistencies
N_2 = number due to incompleteness
N_3 = number due to misinterpretation

4.23.3 Implementation. The implementation of an *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 keywords, phrases, functional and/or performance requirements and expected outputs are documented on decomposition forms (see Fig 4.23.3-1).
(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 examining the system verification diagram and identifying errors as follows:
    (a) Inconsistencies — Decomposition elements that do not accurately reflect the system requirement specification.
    (b) Incompleteness — Decomposition elements that do not completely reflect the system requirement specification.
    (c) 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.

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.

Fig 4.23.3-1
Decomposition Form

STIMULUS	LABEL	RESPONSE
	FUNCTION	
SPEC. REFERENCE		

4.24 Test Coverage

4.24.1 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, in particular, unit, functional, system and acceptance tests. Developers, using the program class of primitives, can apply the measure in unit test to obtain a measure of thoroughness of structural tests. System testers can apply the measure in two ways. First, by focusing on requirements primitives, the system tester can gain a user-view of the thoroughness of functional tests. Second, by focusing on the program class of primitives, the system tester can determine the amount of implementation in the operational environment.

4.24.2 Primitives. The primitives for test coverage are in two classes, program and requirements. 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. Requirements primitives are either test cases or functional capabilities.

4.24.3 Implementation. Test coverage (*TC*) is the percentage of requirements 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:

$$TC(\%) = \frac{\text{(implemented capabilities)}}{\text{(required capabilities)}} \times \frac{\text{(program primitives tested)}}{\text{(total program primitives)}} \times 100$$

4.25 Data or Information Flow Complexity

4.25.1 Application. This is a structural complexity or procedural complexity measure that can be used to evaluate:

(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.

4.25.2 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)

4.25.3 Implementation. Determine the flow of information between modules and/or subsystems by automated data flow techniques, HIPO charts, etc.

A local flow from module *A* to *B* exists if one of the following occurs:

(1) *A* calls *B*,
(2) *B* calls *A* and *A* returns a value to *B* that is passed 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

4.26 Reliability Growth Function

4.26.1 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 achieved. It is an indicator of the level of success of fault correction.

4.26.2 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

4.26.3 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 if 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

$$R(k) = R(u) - A/k \qquad \text{(Eq 4.26-1)}$$

where

$$R(u) = R(k) \text{ as } k \to \infty$$

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 squares estimates. The two equations used to derive the estimates are

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k} \right) = 0 \qquad \text{(Eq 4.26-2)}$$

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k} \right) \frac{1}{k} = 0 \qquad \text{(Eq 4.26-3)}$$

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 Eq 4.26-2 and Eq 4.26-3 to estimate the fault insertion rate, r:

$$\sum_{k=1}^{NS} \left(\frac{nc_k}{NR_k} - R(u) + \frac{A}{k(1-r)} \right) \frac{1}{k(1-r)} = 0 \qquad \text{(Eq 4.26-4)}$$

4.27 Residual Fault Count

4.27.1 Application. This measure provides an indication of software integrity.

4.27.2 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

4.27.3 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.

4.28 Failure Analysis Using Elapsed Time

4.28.1 Application. 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

(1) Predict the total number of faults in the software
(2) Predict the software reliability
(3) Estimate required testing time for reliability goal
(4) Estimate required resource allocation

4.28.2 Primitives

t_i = observed time between failures of a given severity level (for example, execution time)

4.28.3 Implementation. Observed failures should be recorded and the associated faults identified. The time between failures (CPU or other unit that 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 (for example, goodness-of-fit), then the model can be used to estimate the remaining faults (N), MTTF (M), and reliability (R).

4.29 Testing Sufficiency

4.29.1 Application. This measure assesses the sufficiency of software-to-software integration testing by comparing actual to predicted faults.

4.29.2 Primitives

$\hat{NF}$ = 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

4.29.3 Implementation. The fault density (see 4.1) 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

$$NF_{\text{rem}} = (NF - F_{pit})\, M_{it}/M_{\text{tot}}$$

Maximum and minimum tolerance coefficients (l_1 and l_2) should be established based on user experience. Values of 0.5 and 1.5 are suggested for use as the default values.

The guidelines shown in Table 4-1 assume that all scheduled integration tests have been run.

4.30 Mean-Time-to-Failure

4.30.1 Application. This measure is used for hypothesis testing a specified MTTF requirement.

4.30.2 Primitives. Mean-time-to-failure is the basic parameter required by most software reliability models. 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 that might require less resolution, an estimate based on wall-clock time could be used.

4.30.3 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

Table 4-1
Guidelines for Integration Testing Sufficiency

Measure Result	Interpretation	Recommended Action	Comments
$l_1 NF_{\text{rem}} < F_{it} < l_2 NF_{\text{rem}}$	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 Recalculate F_p	Error-ridden modules may exist that should be replaced by a redesigned module
$F_{it} < l_1 NF_{\text{rem}}$	Detected less faults than expected	Apply test coverage measure (see 4.24) before proceeding	May not have designed adequate number or variety of tests

failure analysis must be performed to determine the severity and complexity. Prior failure experience or model fitting analysis (for example, goodness-of-fit test) can be used to select a model representative of a failure process, and to determine a reinsertion rate of faults.

4.31 Failure Rate

4.31.1 Application. This measure can be used to indicate the growth in the software reliability as a function of test time.

4.31.2 Primitives

t_i = observed times between failures (for example, execution time) for a given severity level, i=1, ...

f_i = number of failures of a given severity level in the ith time interval

4.31.3 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 nonhomogeneous Poisson process (NHPP) or a Bayesian type model. The failure rate is

$$\lambda(t) = -\frac{1}{R(t)}\left[\frac{dR(t)}{dt}\right]$$

where

$$R(t) = 1 - F(t)$$

4.32 Software Documentation and Source Listings

4.32.1 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.

4.32.2 Primitive. The questionnaires examine the following primitive product characteristics:

(1) Modularity
(2) Descriptiveness
(3) Consistency
(4) Simplicity
(5) Expandability
(6) Testability

Subcharacteristics include format, interface, math models, and data/control.

4.32.3 Implementation. Two questionnaires, the 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 [2]." The guide can be ordered from the Air Force Operational Test and Evaluation Center.

For the *software documentation evaluation*, the resource documents should include those that contain the 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:

(1) The documentation indicates that data storage locations are not used for more than one type of data structure.
(2) Parameter inputs and outputs for each module are explained in the documentation.
(3) Programming conventions for I/O processing have been established and followed.
(4) The documentation indicates the resource (storage, timing, tape drives, disks, consoles, etc) allocation is fixed throughout program execution.
(5) The documentation indicates that there is a reasonable time margin for each major time-critical program function (rate group, time slice, priority level, etc).
(6) 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:

(1) Each function of this module is an easily recognizable block of code.
(2) The quantity of comments does not detract from the legibility of the source listings.

(3) Mathematical models as described/derived in the documentation correspond to the mathematical equations used in the module's source listing.
(4) Esoteric (clever) programming is avoided in this module.
(5) The size of any data structure that affects the processing logic of this module is parameterized.
(6) Intermediate results within this module can be selectively collected for display without code modification.

4.33 RELY (Required Software Reliability)

4.33.1 Application. RELY provides at the early planning phases of a project a measure that makes visible the trade-offs 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.

4.33.2 Primitives. Required Reliability Rating:

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 mode.[4]

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.[5]

Nominal — The effect of a software failure is a moderate loss to users, but a situation from which one can recover without extreme penalty. Typical examples are management information systems or inventory control systems.[6]

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.[7]

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.[8]

4.33.3 Implementation. Depending on the required reliability 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 4.33.3-1[9] shows the effort factors relative to nominal required to achieve each of the required reliabililty ratings.

This technique can be used conversely to provide a reliability rating for a product through examination of the processes used to develop it (Fig 4.33.3-2).[10]

4.34 Software Release Readiness

4.34.1 Application. This measure is used to quantify the user and supporter (consumer) risk of ownership, based on either quantitative or subjective assessments of user and supporter 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 Operational Test and Evaluation phases, the risk assessment method is applicable during any phase of the life cycle by the identification of appropriate issues and the selection or design of corresponding measures for assessment.

4.34.2 Primitives. The measurements of effectiveness, $ME(i)$, i=1, ..., I, for issues relating to software risk assessment. Example issues are

(1) Functional test coverage
(2) Software maturity
(3) Software source code listings quality factors
(4) Documentation quality factors
(5) Software operator-machine interface
(6) Software support resources assessments

4.34.3 Implementation. This measure assumes that I issues can be combined to assess the readiness of the 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

[4] Definition of primitives and figures from Barry W. Boehm, *Software Engineering Economics*© 1981, pp 374-376. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ.

[5] See footnote 4.

[6] See footnote 4.

[7] See footnote 4.

[8] See footnote 4.

[9] See footnote 4.

[10] See footnote 4.

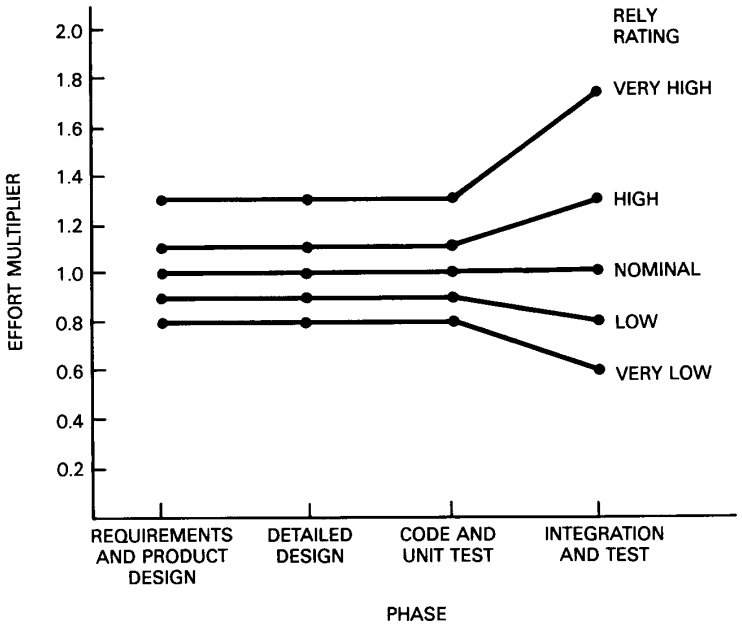


Fig 4.33.3-1
Effort Multipliers by Phase: Required Software Reliability
(Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ.)

factor primitives, the associated risk is determined by

$$RK(i) = 1.0 - ME(i)$$

Furthermore, 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 $W(i)$ fo all user (supporter) measures sum to 1.0. Furthermore, 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 4.10]) and functional test coverage (the fraction of code demonstrated during a test [see 4.5]). 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 [for example, $RK(i) = 1.0 - ME(i)$]. Or for subjective assessments:
    Low risk = 0.0 to 0.33
    Medium risk = 0.34 to 0.66
    High risk = 0.67 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, where zero represents no influence (low correlation) and one represents high dependence (high correlation). Coefficient $S(i,j)$ is the measured or subjective value of measure $ME(i)$'s depen-

Rating	Requirements and Product Design	Detailed Design	Code and Unit Test	Integration and Test
Very high	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

Fig 4.33.3-2
Projected Activity Differences Due to Required Software Reliability
(Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ.)

dence 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:

$$RK'(i) = \frac{\sum_{j=1}^{I} RK(i) \times S(i,j)}{\sum_{j=1}^{I} S(i,j)}$$

Then combined user (supporter) risk is

$$R_{(\text{user/supporter})} = \sum_{i=1}^{I} RK'(i) \times W(i)$$

4.35 Completeness

4.35.1 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.

4.35.2 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, options
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

4.35.3 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

(1) The definitions of the primitives for the particular application must be determined.
(2) 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 points
$D_6 = (B_{12} - B_{11})/B_{12}$ = 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

4.36 Test Accuracy

4.36.1 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.

4.36.2 Primitives

N_s = number of seeded faults
$\hat{N}_s$ = estimated number of detectable seeded faults

4.36.3 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:

$$\text{ALPHA} = \hat{N}_s/N_s$$

4.37 System Performance Reliability

4.37.1 Application. This measure assesses the system's performance reliability, the probability that the value of each performance requirement 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

(1) Reliability planning functions are done by all inter- and intra-organizations.
(2) Periodic reliability estimations during the software life cycle are done to track reliability development.
(3) Verification and validation are done on the system capacity.

4.37.2 Primitives. The following primitives quantify the basic performance attributes that are measured over a time period of length T.

A = number of arrivals (functional jobs) during the time period T
SB = total amount of time "server" is busy
J = number of jobs completed during T
VR = number of requests (visits) per functional job for each server during time period T
T = time period over which measurements are made

4.37.3 Implementation. These measures are applicable to operating systems and networks. Based upon the measured primitives the derived quantifiable measures are

Γ = A/T, average arrival rate of jobs
X = J/T, average throughput rate
U = SB/T, average utilization
AS = SB/J, average service time
WT = $f(\Gamma, U, AS, VR$, waiting time distribution (expressed as a function of arrival rate, utilization, service time, and visit rate)
SE = $A/(AS+W)$, server's efficiency of each job
$R = \sum_{i=1}^{k} (VR \times S)_i + \sum_{i=1}^{k} (VR \times W)_i$, total response time for each functional job over the k servers
AQ = $W \times \Gamma$, the average queue length

(Eq 4.37-1)

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 achieve this.

(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, visit ratios of each system resource are measured for each class of job.
(4) Solve the QNM. 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 by Eq 4.37-1. Calculate the system performance reilability as Reliability = Prob $\{R \leq R_o\}$ for stated assumptions during time period of length T. The system performance is not achieved if the desired reliability is not attained.

4.38 Independent Process Reliability

4.38.1 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.

4.38.2 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

4.38.3 Implementation. This measure assumes that a large program is composed of logically independent modules that 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

$$R = \sum_{VP_i \in L} q_i r_i$$

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 Mar-

kov 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, that is,

$$R = S(1, MD)\, R_n$$

where

$$S = [I-Q]^1$$

Q is the state transition matrix spanning the entire Markov graph program and 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.

4.39 Combined Hardware and Software (System) Operational Availability

4.39.1 Application. This measure provides a direct indication of system dependability, that is, if it is operating or operable when required by the user.

4.39.2 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 ($0 \le P_s \le 1$)
P_h = probability of correctly repairing a hardware failure ($0 \le P_h \le 1$)

4.39.3 Implementation. Tha 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

$$P_{NF,n}(t) = G_{NF,n}(t) - \sum_{j=1}^{K} \cdot \frac{\prod_{i=n+1}^{K} (P_s\lambda_i\mu_i)(-x_jP_h\gamma)^{NF-n} A(-x_j)(1-e^{-x_jt}}{x_j \prod_{\substack{i=1 \\ i\neq j}}^{K} (-x_j + x_i)}$$

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:

$$S^3 + S^2(\lambda_i + \mu_i + \beta + P_h\gamma) + S(P_s\lambda_i\mu_i + \beta\mu_i + \lambda_iP_h\gamma) + \mu_iP_h\gamma) + P_sP_h\gamma\lambda_i\mu_i$$

Also

$$A(-x_j) = (-x_j\beta)(-x_j + \mu_n) + \lambda_n(-x_j + P_s\mu_n)(-x_j + P_h\gamma)$$

Acknowledgment

The following organizations supported the development of this standard:

Aerospace Corporation
Apple Computer, Inc.
Argonne National Laboratory
Bechtel Group, Inc.
Bell Canada
Boeing Aerospace
Computer Sciences Corp.
CTA, Inc.
Digital Equipment Corp.
Dynamics Research Corp.
EG&G
Federal Aviation Administration
General Dynamics
General Electric
Hewlett-Packard
Honeywell
Hughes Aircraft Co.
IBM
INTEL Corporation
International Bureau Software Test
ITT Telecom
Martin-Marietta
McDonnell Douglas
NASA
National Bureau of Standards
National Centre of Systems Reliability
NCR Corporation
Norden Systems
PRC
Product Assurances Consulting
RCA
Sanders Associates
Satellite Business Systems
SOFTECH Microsystems
SoHaR, Incorporated
Sperry
Stromberg Carlson
Tadiran, Ltd.
Tandem Computers
Teledyne Brown Engineering
Transport Canada
TRW
U.S. Air Force
U.S. Army
U.S. Navy
University of Southwestern Louisiana
Xerox Corporation