SAE International

SURFACE VEHICLE/ AEROSPACE RECOMMENDED PRACTICE

SAE JA1003 MAY2012

Issued 2004-01
Reaffirmed 2012-05

Superseding JA1003 JAN2004

Software Reliability Program Implementation Guide

RATIONALE

JA1003 has been reaffirmed to comply with the SAE five-year review policy.

Foreword

In 1994, the SAE G-11 Reliability, Maintainability, Supportability and Logistics (RMSL) Division chartered a software committee, G-11SW, to create several software standards and guidance documents across the RMSL spectrum, including a software reliability program standard and implementation guidelines. The committee was formed as a cross section of international representatives from commercial industries and governments.

The G-11SW committee has developed a standard (JA1002) and these implementation guidelines (JA1003) that are consistent with a SAE G-11 system level reliability program standard (JA1000) and guidelines (JA1000-1), augmented by necessary software-specific information. The G-11SW committee believes these documents reflect the best current commercial practices, and meet the objectives of the United States Department of Defense Acquisition Reform initiative and the North Atlantic Treaty Organization (NATO) Reliability Program. The JA1002 program standard is intended to be used by industries to address market demands for reliable software products that improve system productivity, time to market, and cost-effective implementation. The JA1003 guidelines address possible task activities, methods and techniques through which JA1002 can be implemented. As appropriate, governments and other organizations may also reference these documents.

Software has been recognized by SAE G-11 as an important system component that is not adequately addressed at the system level. Software requires interpretation and variations on RMSL methods used by hardware. In particular, the distinction between hardware and software methods may be difficult to define with such products as programmable logic devices and Field Programmable Gate Arrays. This document provides a selection of methods and techniques to implement the simple concept of supplier-customer-certification authority dialogue and partnership to define, meet, and demonstrate assurance of software product reliability requirements. JA1003 describes how to structure a Plan in terms of activities, tasks, and methods so as to achieve the requirements of JA1002 and provide demonstrated Case evidence of reliability achievement.

Development of these guidelines has required dedication by a few participants and extended review by a wider audience of potential users over an extended period of time. The professionalism of all these individuals and the support they received from their companies, governments, and other organizations is gratefully acknowledged.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE WEB ADDRESS: http://www.sae.org

Abstract

This document defines practices for the implementation of a reliability program for software within an overall systems engineering framework. Guidelines for implementation of a Software Reliability Plan and associated Software Reliability Case are presented. Practices are described for establishing a software reliability program through selection of life cycle activities tailored for the application. Numerous analysis, design, and verification methods and techniques that might be selected to achieve the life cycle activities are summarized and references provided. Guidelines for tailoring a software reliability program include safety and security concerns, integration of Off-The-Shelf software, and collection of appropriate data. The guidelines are applicable to all projects incorporating software, particularly high consequence systems where software reliability is critical. Stakeholders include end-users as well as organizations that acquire, develop or provide post-delivery operation of or support for software.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. Scope

1.1 Purpose

This document provides methods and techniques for implementing a reliability program throughout the full life cycle of a software product, whether the product is considered as standalone or part of a system. This document is the companion to the Software Reliability Program Standard [JA1002]. The Standard describes the requirements of a software reliability program to define, meet, and demonstrate assurance of software product reliability using a Plan-Case framework and implemented within the context of a system application.

This document has general applicability to all sectors of industry and commerce and to all types of equipment whose functionality is to some degree implemented by software components. It is intended to be guidance for business purposes and should be applied when it provides a value-added basis for the business aspects of development, use, and sustainment of software whose reliability is an important performance parameter. Applicability of specific practices will depend on the reliability-significance of the software, application domain, and life cycle stage of the software.

Following guidelines in this document does not guarantee required reliability will be achieved, or that any certification authority will accept the results as sufficient evidence that requisite reliability has been achieved. Following guidelines in this document will provide insight into what level of reliability has been achieved. With proper customer, certification authority, and supplier negotiation and interaction in accordance with these guidelines, it is more likely that the achieved reliability will be acceptable.

1.2 Audience

The target audience for this document includes customer organizations, certification authorities, specialty reliability engineers, and software developers that acquire, develop, use, or provide post-delivery operation of or support for software.

1.3 Applications

The guidance in this document can be applied to all software-intensive projects, and in particular to projects where the reliability of the software is critical to the performance of the system mission. System applications include military, aerospace, transportation, medical, nuclear industries, ground vehicles, and other consumer applications. Such systems may include the integration of custom software as well as Off-The-Shelf (OTS) software. Custom software is generally newly developed software or a significant rework/upgrade of existing software that is for use with a specific application. OTS software sources include commercial vendors, government, and industry. The guidance in this document is generally applicable throughout the complete life cycle, although specific approaches may be more effectively applied at specific life cycle points depending on the software source, application, and pedigree.

1.4 Background

Software is a major component of most important system applications. Because the software component typically provides critical functions, faults in the software may cause the system to fail in a significant way. Such system failures due to direct cause software faults are what we classify as “software failures”. Thus, it is important to use methods and techniques that provide evidence that the software component has been designed, implemented, tested, installed, and, as necessary, updated without faults that might result in undesirable system failures.

The topic of software reliability is concerned with all life cycle activities that prevent, detect, remove, and/or mitigate software faults, and that verify/validate the degree to which software faults do not exist and will not cause system failures. Software reliability is (quantitatively) defined as the probability of failure-free operation of a software program for a specified time under specified conditions. However, having a “number”, even with the appropriate accompanying evidence, is not generally sufficient to convince customers, regulatory authorities, or even the system/software suppliers that the software satisfies its requirements. Thus, software reliability is also (qualitatively) defined as a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. Attributes that relate to implementation of fault tolerance design, use of best practice engineering practices, application of specialized methods and techniques for ensuring safety- and/or security-critical requirements, and procedural methods to ensure mistake-proof loading and/or operation also provide evidence that improves the confidence that the software will not cause a system failure.

There are similarities between hardware and software failures and also differences. Software failures are primarily the result of design defects (during development or maintenance). Other failure sources include use-induced degradation as well as inadequate operational procedures and logistics operations documentation that is considered part of the “software data package”. Hardware failures are primarily the result of physical wear out. Other failure sources include design defects, manufacturing quality deficiency, or maintenance or operating errors. Some system failures are the result of a combination of hardware and software faults. It is generally easier to implement changes to software than to hardware, although any component change must be part of a system support concept that includes continued reliability analysis. Hardware is generally repaired to an original state, unless there is a reason to modify it. Software can frequently be returned to its original state by re-initializing, and often is corrected, enhanced, and adapted so as to become a new version, that is, a new product.

Both hardware and software must be managed as an integrated system. The reliability of the system will depend on the reliability of the hardware and software as an integrated whole. Some techniques to manage the system reliability will be similarly applied to hardware and software components whereas other techniques will be unique to hardware or to software. In addition, the application of a given technique may be different for software than for hardware.

There are no existing methods that guarantee delivered software has no faults. That is, there is always some likelihood that under certain environmental conditions and system operational use, faults in software will be encountered that result in failures of the system. In short, software reliability is not "1.0". There are existing methods and techniques that correlate with delivery of software with reduced faults/failures. It is desirable to provide sufficient quantitative and qualitative evidence that appropriate development and support activities have been conducted to prevent, detect, remove, and/or mitigate possible software faults, particularly those faults that might result in critical system failures.

How might faults be prevented, detected, removed, and/or mitigated in the software development and/or support activities? What techniques might be used to provide quantitative or qualitative evidence that faults capable of causing a system failure do not exist in the software component? Given limited resources and time, which combination of techniques provides the “optimum” cost/benefit results? How are decisions made to select such techniques and how is the evidence from the use of such techniques collected and presented? It is these concerns for which this document provides some guidelines – both for management of a software reliability program and conduct of life cycle activities using appropriate software engineering and reliability-specific techniques.

The reference by Littlewood [LITTLEWD00] describes some of the challenges of providing evidence that can support pre-operational claims for reliability. "Particularly when high levels of reliability need to be assured, it will be necessary to use several sources of evidence to support reliability claims. Combining such disparate evidence to aid decision making is itself a difficult task and a topic of current research." Four areas of evidence are discussed in terms of benefits and limitations:

1. evidence from software components and structure;

2. evidence from static analysis of the software product;

3. evidence from testing of software under operational conditions; and

4. evidence of process quality.

Among the challenges to provide software reliability assurance, there are cultural issues in addition to hard technical research questions to be investigated.

The guidelines in this document recommend determining, meeting, and demonstrating the assurance of customer requirements with an integrated and agreed-upon set of activities and measures within a system context, using a plan-case management framework. It is the hope that these guidelines will be a basis for promoting a systematic approach to the assurance of software reliability through direct attention to the cultural issues of negotiation, implementation agreement, and human interface as well as to the hard technical research necessary to demonstrate progress in understanding this complex area.

1.5 Roadmap to Document Guidance

Each reader of this guide may have different interests. A quick roadmap summary of sections of this guideline that might support reader interests is contained in Table 1.

TABLE 1—DOCUMENT ROADMAP TO TOPICS OF INTEREST

Topic/Question of Interest	Document Section(s) to Read
What are software reliability management concerns across the life cycle?	Section 4
What specific task activities are recommended for a software reliability program?	Section 5
Where is the content of a software reliability plan discussed?	Section 5.1, Appendix B
Where is the content of a software reliability case discussed?	Section 5.2, Appendix B
How do safety and/or security concerns relate to a software reliability program?	Section 6.2
Where are Off-The-Shelf and reused software reliability concerns described?	Section 6.3
What data should be collected as part of a software reliability program?	Section 6.4
Are there any examples that might assist in understanding software reliability?	Appendix D, Appendix E
What analysis, design, and/or verification techniques are available to support software reliability task activities?	Appendix C
What tailoring guidance is provided for a software reliability program?	Section 6.1
What other standards and guidelines exist in this area and how does JA1003 relate to these documents?	Appendix A
What references might be useful to review?	Section 2, Appendix C
Where is the software reliability terminology defined?	Section 3

This guide is not intended to be a novel read from front to back. The life cycle management information described in Section 4 provides an overview of a software reliability program across the various life cycle phases, including example methods/techniques that might support the program in each phase. The task activities described in Section 5 directly support implementation of the software reliability program standard [JA1002] requirements. If the reader is interested in considerations for tailoring a program, addressing safety/security, integrating Off-The-Shelf software, or data collection then Section 6 would be the place to find such information.

The relationship of this software reliability guideline document to many existing standards and guidelines documents is presented as a matrix in Appendix A. Software reliability plan and case outlines are illustrated in Appendix B. If there is interest in a wide variety of potential methods and techniques, then Appendix C would be the place to look. It is emphasized that there are numerous ways to combine and integrate methods and techniques different from those described in Appendix C. There are undoubtedly excellent methods and techniques that exist and are not included in this guide or that are developed after publication of this guide. Numerous tools exist to support the methods and techniques, but this guide does not specifically discuss any of those tools since their capabilities change so rapidly. Such tools can be identified through the many references. Two case studies (at least fragments) are covered in Appendix D and Appendix E. A fairly comprehensive glossary of acronyms and definitions is contained in Section 3 and primary references from SAE, related standards and guidelines, and publications of interest are contained in Section 2. It is also noted that many other references are contained in Appendix C as part of each specific method/technique.

2. References

2.1 Applicable Publications

The indicated references and web links were the current known version as of the publication of this guide. See Appendix C for additional references specific to reliability methods and techniques. There are numerous other publications relevant to software reliability.

2.1.1 SAE PUBLICATIONS

Available from SAE, 400 Commonwealth Drive, Warrendale, PA 15096-0001.

AIR5022—Reliability and Safety Process Integration
ARP5580—Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications
J1739—Potential Failure Mode and Effects Analysis in Design (Design FMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) and Effects Analysis for Machinery (Machinery FMEA)
JA1000—Reliability Program Standard
JA1000-1—Reliability Program Implementation Guide
JA1002—Software Reliability Program Standard
JA1004—Software Supportability Program Standard
JA1005—Software Supportability Program Implementation Guidelines
JA1006—Software Support Concept
JA1010—Maintainability Program Standard
JA1010-1—Maintainability Program Implementation Guide

2.2 Related Standards

A world-wide search capability for reliability standards and standards developing organizations is available from: IIT Research Institute / Reliability Analysis Center, 201 Mill Street, Rome, NY 13440-6916

2.2.1 AIAA PUBLICATIONS

Available from American Institute of Aeronautics and Astronautics (AIAA), 1801 Alexander Bell Drive, Suite 500, Reston, VA 20191-4344.

AIAAR013—ANSI/AIAA R-013-1992, “AIAA Recommended Practice for Software Reliability,” February 1993.

2.2.2 BRITISH STANDARDS INSTITUTE PUBLICATIONS

Available from British Standards Institute (BSI), Linford Wood Milton Keyes, MK14 6LE UK.

BS5760-P8—BS 5760, “Reliability of Systems, Equipment and Components,” Part 8: “Guide to Assessment of Reliability of Systems Containing Software,” British Standards Institute, Draft for Approval for Publication, July 7, 1997.

2.2.3 DOD PUBLICATIONS

Available from Chief, Bibliographic Systems, U.S. Government Printing Office, Sales Management Division (SSMB), Washington, DC 20402.

MILSTD882D—MIL-STD-882D, "Department of Defense Standard Practice for System Safety," Department of Defense, February 10, 2000.

NUREG6421—NUREG/CR-6421, "A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications," Office of Nuclear Reactor Regulation, US Regulatory Commission, March 1996.

2.2.4 IEC PUBLICATIONS

Available from International Electrotechnical Commission, 1327 Jones Dr., Ann Arbor, MI, 48105.

IEC61508—ISO/IEC 61508, Edition 1.0: "Functional safety—Safety instrumented systems for the process industry sector - Part 1: Framework, definitions, system, hardware and software requirements,” Multi-part standard, International Electrotechnical Commission, 1998.

IEC61511-1—ISO/IEC 61511-1, Edition 1.0: "Functional safety of electrical/electronic/programmable electronic safety-related systems," International Electrotechnical Commission, 2003.

IEC61713—ISO/IEC 61713, Edition 1.0: "Software dependability through the software life-cycle processes - Application guide," International Electrotechnical Commission, June 30, 2000.

IEC61719—ISO/IEC 61719 (Draft): "Guide to measures to be used for the quantitative dependability assessment of software," ISO/IEC/TC56/SC7/WG10/N111, Draft February 11, 2000.

2.2.5 IEEE PUBLICATIONS

Available from IEEE Computer Society, Publications Office, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, CA 90720-1264.

IEEE12207-0—IEEE/EIA Std 12207.0-1996, “Software life cycle processes,” IEEE Computer Society, March 1998.

IEEE12207-1—IEEE/EIA Std 12207.1-1997, “Software life cycle processes - Life cycle data,” IEEE Computer Society, April 1998.

IEEE12207-2—IEEE/EIA Std 12207.2-1997, “Software life cycle processes – Implementation considerations,” IEEE Computer Society, April 1998.
IEEE610—IEEE Std-610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Computer Society, September 1990.
IEEE982-1—IEEE Std-982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Computer Society, June 1988.
IEEE982-2—IEEE Std-982.2-1988, "IEEE Guide for the use of Standard Dictionary of Measures to Produce Reliable Software,” IEEE Computer Society, September 1988.
IEEE1028—IEEE Std-1028-1994, "IEEE Standard for Software Reviews," IEEE Computer Society, December 1997.
IEEE1220—IEEE Std-1220-1998, "IEEE Standard for Application and Management of the Systems Engineering Process," IEEE Computer Society, December 1998.
IEEE1228—IEEE Std-1228-1994,"IEEE Standard for Software Safety Plans," IEEE Computer Society, March 1994.
IEEE1413—IEEE Std-1413-1998,"IEEE Standard Methodology for Reliability Prediction and Assessment for Electronic Systems and Equipment," IEEE Reliability Society, December 1998.

2.2.6 ISO PUBLICATIONS

Available from Europe: ILI, Index House, Ascot, Berkshire, SL5 7EU, UK, USA: ILI, 610 Winters Avenue, Paramus, NJ 07652, USA, Germany: ILI, Dietlindenstraße 15, D-80802, Munich, Deutschland, Italy: ILI, Via Guido D'Arezzo, 4 - 20145 Milano, France: ILI, 25 rue de Ponthieu, 75008 Paris, France.

ISO12207—ISO/IEC 12207, “Software Life Cycle Processes,” August 1, 1995.
ISO15288—ISO/IEC 15288, “Systems Engineering – System Life Cycle Processes,” Edition 1, November 8, 2002.

2.2.7 MISRA PUBLICATIONS

Available from Motor Industry Software Reliability Association (MISRA), Electrical Group, MIRA Ltd, Watling Street, Nuneaton, Warwickshire CV10 0TU UK.

MISRA-VBS—ISO/TR 15497, "Development Guidelines for Vehicle Based Software, the Motor Industry," Motor Industry Software Reliability Association, ISBN 0 9524156 0 7, November 1994.

2.2.8 NATO PUBLICATIONS

Available from Directorate of Standardization, Stan 2, Kentigern House, 65 Brown Street, GLASGOW G2 8EX, United Kingdom.

ARMP1—ARMP-1, Edition 3, "NATO Requirements for Reliability and Maintainability," June 2002.
ARMP4—ARMP-4, Edition 2, "Guidance on Writing NATO R&M Requirements Documents," October 2001.
ARMP6—ARMP-6, Edition 1, "Monitoring and Managing In-Service R&M," December 1988.
ARMP7—ARMP-7, Edition 1, " NATO R&M Terminology Applicable to ARMPs," July 2001.
NATO96—NATO (Draft), "COTS Software Acquisition Guidelines and COTS Policy Issues—1st Revision,” NATO Communications and Information Systems Agency, January 12, 1996.
NATO97—NATO (Draft), "NATO Guidelines for the Integration of Off-The-Shelf Software," Working Paper AC/322(SC/5)WP/4, NATO C3 Board Information Systems Sub-Committee, June 30, 1997.

2.2.9 NIST PUBLICATIONS

Available from National Institute of Standards and Technology, 100 Bureau Drive, Stop 3460, Gaithersburg, MD 20899-3460.

NIST800-14—NIST 800-14, "Generally Accepted Principles and Practices for Securing Information Technology Systems," National Institute for Standards and Technology, 1996.
NIST800-26—NIST 800-26, "Security Self-Assessment Guide for Information Technology Systems," National Institute for Standards and Technology, 2001.
NIST800-27—NIST 800-27, "Engineering Principles for Information Technology Security (A Baseline for Achieving Security)," National Institute for Standards and Technology, 2001.

2.2.10 RTCA PUBLICATIONS

Available from RTCA, Inc., 1828 L Street, NW, Suite 805, Washington, DC 20036.

DO178B—RCTA/DO-178B/ED-12B, “Software Considerations in Airborne Systems and Equipment,” Federal Aviation Administration software standard, RTCA Inc., December 1992.
DO248B—RCTA/DO-248, Final Report for Clarification of DO-178B, "Software Considerations in Airborne Systems and Equipment,” Prepared by SC-190, October 12, 2001.

2.2.11 SOFTWARE ENGINEERING INSTITUTE PUBLICATIONS

Available from Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

CMMI2000—CMMI-SE/SW-Continuous, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Continuous Representation," CMU/SEI-2000-TR-019, November 2000.
CMMI-SE/SW-Staged, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Staged Representation," CMU/SEI-2000-TR-018, November 2000.
SPICE98—ISO/IEC 15504:1998: “Software Process Improvement Capability Determination (SPICE)—Software Process Assessment,” ISO/IEC/JTC1/SC7/WG10/N111, ISO 1998.

2.2.12 UK MINISTRY OF DEFENCE PUBLICATIONS

Available from UK Defence Standardization, Room 1138, Kentigern House, 65 Brown Street, GLASGOW G2 8EX, United Kingdom.

DEFS0042-2—Defence Standard 00-42 (PART 2)/Issue 1, “Reliability And Maintainability Assurance Guides, Part 2: Software,” United Kingdom Ministry of Defence, September 1997.
DEFS0042-3—Defence Standard 00-42 (PART 3)/Issue 1, “Reliability And Maintainability Assurance Guides, Part 3: R&M Case,” United Kingdom Ministry of Defence, October 1999.
DEFS0055—Defence Standard 00-55 Issue 2, “Requirements for Safety Related Software in Defence Equipment," Part 1: Requirements, Part 2: Guidance," United Kingdom Ministry of Defence, August 1997.
DEFS0060—Defence Standard 00-60, “Integrated Logistic Support”, Issue 2, “Logistic Support Analysis Application to Software Aspects of Systems”, Part 3, United Kingdom Ministry of Defence, March 1998.

2.3 Publications

BASILI02—Basili, Vic, Boehm, Barry, and others, "What We Have Learned About Fighting Defects," Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS™02), IEEE Computer Society, 2002. http://www.CeBASE.org

DACS02—DACS CD, "Software Reliability Source Book," Data and Analysis Center for Software, Rome, NY, 2002. http://iac.dtic.mil/dacs/

FALLA96—Falla, Mike, "Results and Achievements from the DTI/EPSRC R&D Programme in Safety Critical Systems," Edited by Mike Falla, Motor Industry Software Reliability Association, November 1996. http://www.comp.lancs.ac.uk/computing/resources/scs/

HELAN98—Helander, M., Shao, M., and Ohlsson, N. "Planning Models for Software Reliability and Cost," IEEE Transactions on Software Engineering, Vol 24, Number 6, June 1998, pp 420-434.

HERRM99—Herrmann, D., Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors, IEEE Computer Society, Los Alamitos, CA, 1999.

JONES97—Jones, Capers, "Software Quality In 1997: What Works and What Doesn't," Software Productivity Research, 1997. http://www.spr.com/

LAKEY97—Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

LEVESON95—Leveson, Nancy G., Safeware: System Safety and Computers, Addison Wesley Publishing Company, 1995.

LITTLEWD00—Littlewood, B. and Strigini, L., "Software Reliability and Dependability: a Roadmap, in The Future of Software Engineering", State of the Art Reports given at the 22nd Int. Conf. on Software Engineering, Limerick, June 2000, (A. Finkelstein, Ed.), pp. 177-188, ACM Press, 2000.

LYU96—Lyu, Michael, Handbook of Software Reliability Engineering, ISBN 0-07-039400-8, McGraw Hill, 1996.

MUSA92—Musa, John D. "Operational Profiles in Software Reliability Engineering," IEEE Software, March 1993, pages 14-32.

MUSA99—Musa, John D., Software Reliability Engineering, McGraw-Hill Book Company, NY, 1999.

NEUF02—Neufelder-Owner, A., N., "The Facts About Predicting Software Defects and Reliability," Journal of the RAC, 2ndQ, 2002, pp 1-4.

PRIM97—PRIM-97, "Worldwide Reliability & Maintainability Standards," Reliability Analysis Center, IIT Research Institute / Reliability Analysis Center, Rome, NY, 1997.

SCHN97—Schneidewind, N., "Reliability Modeling for Safety-Critical Software," IEEE Transactions on Reliability, Vol 46, Number 1, March 1997, pp 88-98.

SSSHDBK99—Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

XTALK03—CrossTalk, "Programming Languages," Journal of Defense Software Engineering, Vol. 16 No. 2, February 2003.

3. Definitions

3.1 Acronyms

AIAA	American Institute of Aeronautics and Astronautics
AIR	Aerospace Information Report
ANSI	American National Standards Institute
ARMP	Allied Reliability and Maintainability Publication
ASIC	Application Specific Integrated Circuit
BSI	British Standards Institute
CM	Configuration Management
CMMI	Capability Maturity Model Integrated
COTS	Commercial Off-The-Shelf
DACS	Data and Analysis Center for Software
DoD	Department of Defense
DOE	Design Of Experiment
DSI	Delivered Source Instructions
EIA	Electronics Industries Alliance
FAA	Federal Aviation Administration
FIR	Formal In-Process Review
FMECA	Failure Modes, Effects and Criticality Analysis
FRACAS	Failure Reporting and Corrective Action System
FTA	Fault Tree Analysis
GQM	Goal, Question, Metric
GUI	Graphical User Interface
HCI	Human Computer Interface
I4	Independence, Isolation, Interoperability, Incompatibility
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
JA	Two character code for SAE ground vehicle (J) and aerospace (A) standards and guidelines
KSLOC	Thousands (K) of Source Lines of Code
MISRA	Motor Industry Software Reliability Association
MOD	Ministry Of Defence (United Kingdom)
NASA	National Aeronautics and Space Administration
NATO	North Atlantic Treaty Organization
NCSLOC	Non-Commented Source Lines of Code
NDI	Non-Developmental Item
NIST	National Institute of Standards and Technology
OTS	Off-The-Shelf
QA	Quality Assurance

QFD	Quality Function Deployment
R&M	Reliability and Maintainability
RAC	Reliability Analysis Center
RMSL	Reliability, Maintainability, Supportability, Logistics
SAE	Society of Automotive Engineers
SEI	Software Engineering Institute
SFMECA	Software FMECA
SFTA	Software FTA
SRE	Software Reliability Engineering
UK	United Kingdom
V&V	Verification and Validation

3.2 Terms

The following key terms are defined. Reference [IEEE610] is a generally applicable reference for software terms not defined in this section. Some terms apply only to software, but many terms apply more generally to a system of which software is a component. The specific source of the definitions in this section is referenced as follows:

[0] term defined by its use in this guide
[1] reference [AIAAR013]
[2] reference [JA1002]
[3] reference [JA1000-1]
[4] reference [ISO12207]
[5] reference [MUSA99]
[6] reference [JA1005]
[7] reference [DO178B]
[8] reference [MIL-STD-882D]

3.2.1 ACQUIRER[4]

An organization that procures a system, software product or software service from a supplier.

NOTE—The acquirer could be one or more of the following: buyer, owner, user, and/or purchaser.

3.2.2 CERTIFICATION[7]

Legal recognition by the certification authority that a product, service, organization or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval or other documents as required by national laws and procedures.

3.2.3 CERTIFICATION AUTHORITY/REGULATOR [7]

The organization or person responsible within the state or country concerned with the certification of compliance with the requirements.

3.2.4 CONTRACT[4]

A binding agreement between two parties, especially enforceable by law, or a similar internal agreement wholly within an organization, for the supply of software service or for the supply, development, production, operation, or maintenance of a software product.

3.2.5 COVERAGE[0]

The ratio of actual to possible software features/functions, requirements, statements, and/or branches/paths that are exercised during one or more test cases. Types of coverage can be categorized by the unit, e.g., feature coverage, requirements coverage, statement coverage, path coverage.

3.2.6 CUSTOMER[0]

See Acquirer.

3.2.7 DEFECT

See Software Defect.

3.2.8 DEPENDABILITY[0]

See Surety.

3.2.9 DESIGN RELIABILITY[0]

(1) The set of activities that focus on the prevention, detection, prediction, estimation, and/or mitigation of defects in software specifications (e.g., user guide, requirements, design, code, test plan/cases). (2) A measure of the remaining defects in software specifications at a specific reference point. (3) A measure of the predicted software reliability at a specific reference point.

3.2.10 DEVELOPER[4]

An organization that performs development activities (including requirements analysis, design, testing through acceptance) during the software life cycle process.

3.2.11 ERROR[1]

(1) A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action that results in software containing a fault.

3.2.12 FAILURE[1]

(1) The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of a functional unit to perform its required function. (3) A departure of program operation from program requirements.

3.2.13 FAILURE INTENSITY[5]

See Failure Rate.

3.2.14 FAILURE MODES, EFFECTS AND CRITICALITY ANALYSIS [3]

A proactive approach used for determining the potential failure modes of a system/equipment (including software), all likely ways in which a component or equipment can fail, causes for each failure mode, and effects/criticality of each failure mode.

3.2.15 FAILURE RATE[1]

(1) The ratio of the number of failures of a given category or severity to a given period of time; for example failures per second of execution time, failures per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a per unit of time, failures per number of transactions, failures per number of computer runs.

3.2.16 FAILURE REPORTING AND CORRECTIVE ACTION SYSTEM [3]

A set of processes, procedures, and tools for reporting, reviewing, analyzing, correcting, and storing information about system/software failures.

3.2.17 FAILURE SEVERITY[ADAPTED FROM 1]

A rating system for the impact of every recognized credible failure mode.

3.2.18 FAULT

See Software Fault.

3.2.19 FAULT TOLERANCE[1]

The survival attribute of a system that allows it to deliver the required service after faults have manifested themselves within the system.

3.2.20 FAULT TREE ANALYSIS [3]

An analysis technique where identified potential system failure modes are analyzed in terms of what potential software faults (single point of failure) or multiple faults (multiple points of failure) might result in the potential failure mode.

3.2.21 I4- INDEPENDENCE, ISOLATION, INOPERABILITY, INCOMPATIBILITY[0]

3.2.21.1 Independence

Multiple, independent subsystems and completely different sources of enabling stimuli for critical functions are incorporated within the system.

3.2.21.2 Isolation

Critical functions are encapsulated separate from any other functions that might cause undefined interactions with the critical functions

3.2.21.3 Inoperability

Critical functions become predictably and irreversibly inoperable in credible abnormal operating environments before the isolation features are compromised

3.2.21.4 Incompatibility

Functional interfaces are constructed so that they are incompatible with functions (in particular safety critical functions) with which they are not intended to interface

3.2.22 LIFE CYCLE MODEL[4]

A framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use.

3.2.23 NON-DELIVERABLE ITEM[4]

Hardware or software product that is not required to be delivered under the contract but may be employed in the development of a software product.

3.2.24 OFF-THE-SHELF PRODUCT[4]

Product that is already developed and available, usable either "as is" or with modification.

3.2.25 OPERATIONAL PROFILE[5]

The complete set of operations (major system logical tasks) with their probabilities of occurrence.

3.2.26 OPERATIONAL RELIABILITY[0]

(1) The set of dynamic test activities that focus on the prevention, detection, prediction, estimation, and/or mitigation of defects in the operational software code through dynamic unit, integration, acceptance, certification testing). (2) A measure of the remaining faults in software code at a specific reference point. (3) A measure of the estimated software reliability at a specific reference point.

3.2.27 PROCESS[4]

A set of interrelated activities, which transform inputs into outputs.

NOTE—The term "activities" covers use of resources.

3.2.28 QUALIFICATION[4]

The process of demonstrating whether an entity is capable of fulfilling specified requirements.

3.2.29 QUALITY ASSURANCE[4]

All the planned and systematic activities implemented within the quality system, and demonstrated as needed, to provide adequate confidence that an entity will fulfill requirements for quality.

NOTES—(1) There are both internal and external purposes for quality assurance: (a) Internal quality assurance: within an organization, quality assurance provides confidence to management; (b) External quality assurance: in contractual situations, quality assurance provides confidence to the customer or others. (2) Some quality control and quality assurance actions are interrelated. (3) Unless requirements for quality fully reflect the needs of the user, quality assurance may not provide adequate confidence.

3.2.30 SAFETY[8]

Freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.

3.2.31 SECURITY[0]

Features and procedures of a system that ensure its requirements are met for timely access to authenticated services and for protection from denial of authenticated services.

3.2.32 SOFTWARE DEFECT[0]

Any condition in a software artifact (e.g., specification, code, test) that if left unchanged could result in a software failure. Defect and fault are sometimes considered to be synonymous although fault is more strictly considered to be a defect in the code.

3.2.33 SOFTWARE FAILURE[2]

The inability of a software component to perform its required functions within specified performance requirements.

3.2.34 SOFTWARE FAULT[1]

(1) A defect in the code that can be the cause of one or more failures. (2) An accidental condition that causes a functional unit to fail to perform its required function. Synonymous with bug.

3.2.35 SOFTWARE FAULT[2]

An accidental condition that causes a software functional unit to fail to perform its required function.

3.2.36 SOFTWARE FAULT DENSITY[0]

The ratio of code faults to a unit of size, such as function points, modules, source lines of code at a specific reference point of time, such as at the start of system test or operational use.

3.2.37 SOFTWARE MAINTAINABILITY[6]

The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. Also, a set of attributes that bear on the effort needed to make specified modifications.

3.2.38 SOFTWARE MAINTENANCE[6]

The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

3.2.39 SOFTWARE MODIFICATION SUPPORT[6]

The software support activities of change analysis, implementation, test and release of software products. Changes may be termed corrective, perfective and adaptive, and may also embrace modifications that are designed to prevent foreseeable future software operating problems.

3.2.40 SOFTWARE PRODUCT[1]

The set of computer programs, procedures, and possibly associated documentation and data.

3.2.41 SOFTWARE RELIABILITY[2]

(1) The probability of failure-free operation of a software program for a specified time under specified conditions. (2) A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

NOTE—Quantitative descriptions of software reliability are typically not sufficient evidence that software is reliable, and other evidence such as fault tolerant design, coverage measures, engineering practices, safety analyses, and so forth are necessary to provide adequate confidence that software can be reliably loaded, operated, and supported. Hence, the combination of evidence provided from both definitions (1) and (2) above is typically used to determine the reliability of software.

3.2.42 SOFTWARE RELIABILITY CASE[0]

The evidence presented throughout the project that software reliability requirements are consistent with system level requirements, are achievable, are understood by the development organization, and that ambiguities have been resolved.

3.2.43 SOFTWARE RELIABILITY ENGINEERING[1]

The application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems.

3.2.44 SOFTWARE RELIABILITY ESTIMATION[1]

The application of statistical techniques to observed failure data collected during system testing and operation to assess the reliability of the software.

3.2.45 SOFTWARE RELIABILITY MANAGEMENT[0]

The process of optimizing the reliability of software across the complete software life cycle by emphasizing human error prevention, fault detection and removal, use of measurements to improve reliability, and balancing the level of reliability consistent with project constraints such as resources, schedule, and performance.

3.2.46 SOFTWARE RELIABILITY MODEL[1]

A mathematical expression that specifies the general form of the software failure process as a function of factors such as fault introduction, fault removal and the operational environment.

3.2.47 SOFTWARE RELIABILITY PLAN[0]

A description of the set of activities that will be performed throughout a project to ensure that requirements for software reliability have been defined through negotiations with the customer, analyses have been identified and conducted that ensure customer reliability requirements are met, and demonstrated evidence is provided that the customer reliability requirements have been achieved.

3.2.48 SOFTWARE RELIABILITY PREDICTION[1]

A forecast of the reliability of the software based on parameters associated with the software product and its development environment.

3.2.49 SOFTWARE RELIABILITY PROGRAM[0]

The management infrastructure and activities necessary to adequately integrate software reliability within a system reliability program and provide adequate evidence that the software reliability requirements have been determined, met, and demonstrated. The two key components of the management infrastructure are the Software Reliability Plan and Software Reliability Case.

3.2.50 SOFTWARE RELIABILITY PROGRAM [ADAPTED FROM 3]

The organizational processes and practices that are intended to: (1) Ensure the delivery of a software product that has been adequately designed to achieve its performance specifications within its system application context; and (2) Ensure there is adequate evidence that the performance specification for the delivered software product has been achieved and continues to be met during operational use.

3.2.51 SOFTWARE SAFETY[0]

Features and procedures which ensure that a software product performs predictably under normal and abnormal conditions, thereby minimizing the likelihood of an unplanned event occurring, controlling and containing its consequences, and preventing accidental injury, death, destruction of property and/or damage to the environment, whether intentional or unintentional.

3.2.52 SOFTWARE SECURITY[0]

Features and procedures of the software component of a system that ensure the system's requirements are met for timely access to authenticated services and for protection from denial of authenticated services.

3.2.53 SUPPLIER[4]

An organization that enters into a contract with the customer for the supply of a system, software product or software service under the terms of the contract.

NOTES—(1) The term "supplier" is synonymous with contractor, producer, seller, or vendor. (2) The customer may designate a part of its organization as supplier.

3.2.54 SURETY[0]

Attributes of and activities associated with achieving and assessing system safety, security, and reliability.

3.2.55 SYSTEM[1]

(1) A collection of people, machines and methods organized to accomplish a set of specific functions. (2) An integrated whole that is composed of diverse, interacting, specialized structures and subfunctions. (3) A group or subsystem united by some interaction or interdependence, performing many duties but functioning as a single unit.

3.2.56 SYSTEM RELIABILITY[3]

The ability of a system to perform a stated function under stated conditions, for a stated period of time.

3.2.57 SYSTEM SAFETY[8]

The application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle.

3.2.58 TIME[1]

There are several categories of time that may be of interest for determining when failures occur and the impact of the frequency of the failures. These categories include: (1) Calendar Time: chronological time, including time during which a computer may not be running. (2) Clock Time: elapsed wall clock time from the start of program execution to the end of program execution. (3) Execution Time: the amount of actual processor time used in executing a program.

3.2.59 VALIDATION[4]

Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled.

NOTES—(1) In design and development, validation concerns the process of examining a product to determine conformity with user needs. (2) Validation is normally performed on the final product under defined operating conditions. It may be necessary in earlier stages. (3) "Validated" is used to designate the corresponding status. (4) Multiple validations may be carried out if there are different intended uses.

3.2.60 VERIFICATION[4]

Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.

NOTES—(1) In design and development, verification concerns the process of examining the result of a given activity to determine conformity with the stated requirement for that activity. (2) "Verified" is used to designate the corresponding status.

4. Life Cycle Management

The software reliability program is most effective within a complete life cycle management approach as illustrated in Figure 1. Although it is recommended to initiate a program early in the life cycle, one can be established at any point in the life cycle. It may be more difficult to start a program during later system stages, and although certain practices may not be appropriate there are always some practices that will be effective.

Management and technical activities that might be part of a software reliability program are summarized within this section and specific program tasks are described in Section 5. A comprehensive set of potential analysis, design, and verification methods and techniques that can be applied in various life cycle phases are summarized in Appendix C. Some special software reliability considerations that may need to be integrated into a software reliability program are discussed in Section 6.

4.1 Program Management

The use of a software reliability plan/case framework to capture activity planning and results is the recommended reliability program management mechanism. A software reliability program should be implemented within the context of the operational system in which the software is to operate. In addition, such a program must consider the overlap of requirements that may arise from closely related specialty disciplines, such as safety, security, and supportability, as well as quality engineering in general. Since software reliability is a measure of meeting requirements, this measure should include requirements that derive from all concerns. It is essential that the definition of what constitutes failure is clearly stated and related to requirements.

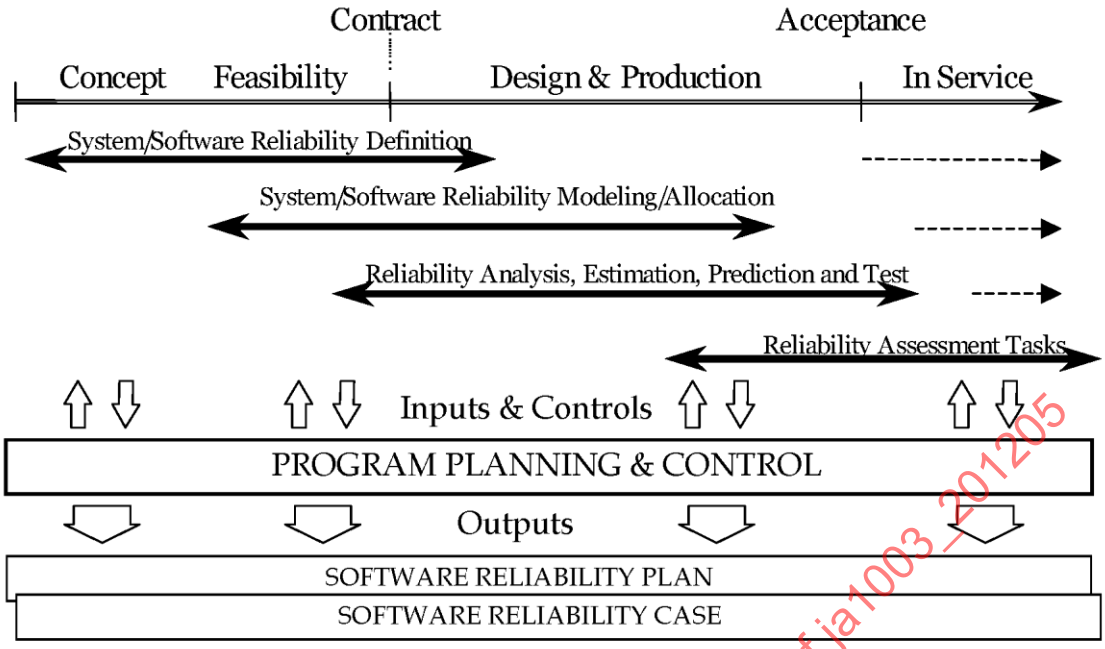

FIGURE 1—SYSTEM/SOFTWARE RELIABILITY LIFE CYCLE MANAGEMENT

Achieving the required reliability for any particular software product will depend on having an effective strategy of planned activities. The strategy should be developed and executed in conjunction with overall system reliability plans. There should be visibility to project and higher program managers of major milestones and outputs to ensure that control and monitoring of reliability activity is integrated with other project activities.

A software reliability plan documents the strategy for satisfying customer reliability requirements and expectations for a software product. This "plan" can assume a variety of physical forms (e.g., separate document, integrated part of other documents, activities integrated into a system Work Breakdown Structure). A software reliability plan needs to address the software aspects of a system reliability plan and describe the specific software-related life cycle activities that are to be undertaken to achieve and sustain the software reliability objectives. Each planned activity should describe associated tasks, schedules, resources, expected methods/techniques to accomplish the activity, and specific metrics to show progress toward goals throughout the life cycle. Results of the activities are described in the software reliability case.

Some of the applicable management practices include the use of life cycle software engineering standards (C.4.3), requirements management (C.4.5), risk management (C.4.6), configuration management (C.4.1), and FRACAS (C.4.2) within the overall system/software project management.

4.2 Technical Activities

Engineering methods can provide confidence that potential faults have been prevented, detected, removed, or mitigated to an acceptable level of confidence. Prototype experiments and trade-offs may be conducted to determine operational profiles and preferred design solutions, depending on the negotiated agreement between the supplier and customer. Software reliability requirements are allocated from system reliability requirements. Software reliability computations are integrated as part of the system reliability computations.

When security, safety, or other high criticality requirements are identified, then specific additional methods can and should be conducted to provide necessary reliability evidence. Depending on the program, this evidence may be separated into distinct "case" reports for reliability, safety, security and so forth, or combined into one "case" report. Thus functional requirements and requirements that derive from specialty interests, as well as selected system- and software-specific methods/techniques determine the context in which a software reliability technical activities are conducted.

In addition, there should be feedback of lessons learned and case evidence from such software reliability activities to enable process improvement. Such feedback should be stored in an organization's overall reliability data base. Thus, each software product's reliability plan and case information can be used in a broader context to support an organization's software reliability process improvement program.

The general criteria for selection of a methods to support technical activities include:

1. substantive use;

2. correlation with technical measurement of defect/failure reduction; and

3. evidence of good benefit to cost performance.

These criteria are typically based on rather qualitative judgment. It is important to understand where defects/faults typically occur in software products as well as the types of defects so specifically effective technical activities can be selected for use. Each application, with customer-/supplier-specific influences, will tend to have unique defect injection/detection/removal profiles. A general view from reference [JONES97] of this type of information for leading, average, and lagging projects is illustrated in Table 2. In addition, defect potential, defect removal efficiency, and delivered defects for a variety of application areas are illustrated in Table 3.

NOTE—Data is in defects per function point.

TABLE 2—EXAMPLE DEFECTS AND REMOVAL EFFICIENCY PROFILES

Task	Leading Organization	Average Organization	Lagging Organization
Requirements	0.55	1.00	1.45
Design	0.75	1.25	1.90
Coding	1.00	1.75	2.35
User manuals	0.40	0.60	0.75
Bad fixes	0.10	0.40	0.85
Total Potential Defect Density	**2.80**	**5.00**	**7.30**
Defect Removal %	95%	85%	75%
Actual Delivered Defect Density (Defects/Function Point)	0.14	0.75	1.83

TABLE 3—EXAMPLE DEFECT REMOVAL EFFICIENCY, DELIVERED DEFECTS FOR APPLICATIONS

	System Software	Commercial Software	Information Software	Military Software	Overall Average
Defect Potentials	6.0	5.0	4.5	7.0	5.6
Defect Removal Efficiency	94%	90%	73%	96%	88%
Delivered Defect Density (Defects/Function Point)	0.4	0.5	1.2	0.3	0.65
First Year Discovery Rate	65%	70%	30%	75%	60%
First Year Defect Density (Defect/Function Point)	0.26	0.35	0.36	0.23	0.30

The following general activities from reference [NEUF02] as illustrated in Table 4 have been correlated with a higher reduction in defects.

TABLE 4—PRACTICES ASSOCIATED WITH REDUCED DEFECT DENSITY

Practice	System Phase & SW Activity
All requirements are mapped to system tests	Concept/Feasibility (SW Requirements)
Requirements are reviewed before designing or coding	Concept/Feasibility (SW Requirements)
System test beds are used	Design&Production (SW Testing)
Test plan started at least one phase of the life cycle before testing begins	Phases before testing (SW Testing)
Testers use a FRACAS (defect tracking system) to determine what to test/retest	Design&Production (SW Testing)
All upgrades made after a system test are regression tested	In Service (SW Support)
Corrective action releases per year <= 4	In Service (SW Support)
All modifications made after a system test are regression tested	In Service (SW Support)
FRACAS used for tracking all corrective actions	In Service (SW Support)
Walk-thrus are performed for all phases of life cycle	All phases

The type of information provided in Tables 2, 3, and 4 can provide guidance to a software project for the effective selection of software engineering and reliability techniques.

Some general areas of technical activity and methods to support that activity throughout the typical life cycle activities are summarized below.

a. Negotiate Reliability Requirements

Reliability requirements are negotiated among supplier, customer and certification authority representatives. The requirements for software reliability contribution to overall system reliability is derived from an understanding of the system reliability goals and objectives. An overall strategy for software reliability achievement is determined including pre-development, development, and in-service concepts, activities, methods and tools, risk analysis, and training and integration with other disciplines and staff. Activity-associated measures are established in terms of goals, assumptions and claims, and expected evidence to meet the claims. It is expected that the reliability requirements may continue to be negotiated somewhat throughout the life cycle.

Example Methods: Quality function deployment, customer surveys, Goal-Question-Metric, operational profiles/scenarios, support scenarios.

b. Develop An Architecture/Design for Reliability

Design the software architecture and implementation to support design requirements for software reliability, and iterate with the system and software design effort to ensure reliability characteristics are built into the software artifacts. Ensure the design and implementation meet specified reliability claims through appropriate defect prevention, detection, removal, and mitigation activities.

Example Methods: Software Reliability Engineering, FMEA, FTA, hazard analysis, design by contract, partitioning, formal specifications/analysis, formal scenario analysis/operational profiles, reliability predictions.

c. Construct a Test Approach as Early as Possible

Test planning should be conducted at least one phase prior to test execution. Reliability engineered testing is planned as part of the test planning approach. Testing is critical providing quantitative evidence that reliability requirements have been met. Ensure testing is designed to meet expected reliability claims, including measurement and categorization of defects found and associated measures of reliability growth.

Example Methods: Test plan/harness developed during design; coverage analysis; reliability predictions, reliability estimation.

d. Apply a Formal Review Process During All Phases of Life Cycle

A formal review process should be applied during all phases of life cycle with the collection of defect data and follow-up review to ensure corrections have been made.

Example Methods: Formal in-process reviews: defects (type and severity, time, source); defect removal efficiency; reliability predictions.

e. Conduct Comprehensive Tests

Testing is performed at all levels (unit, integration, system, alpha/beta) to ensure coverage of requirements. Test cases as determined from the test planning are executed, data collected on failures, continued testing determined through reliability estimation and programmatic cost/schedule constraints. Regression testing is conducted after any modifications due to development system testing or in-service support changes.

Example Methods: Test suites: unit, integration, system; Dynamic testing: functional, interface, performance, probabilistic, regression, stress, usability; equivalent classes to determine prioritization; reliability estimation from test data.

f. Use Failure Reporting and Corrective Action Procedures

Failure reporting and corrective action procedures should be used during development initiate the defect tracking process and to determine testing/retesting activities. This activity is continued during in-service support as part of the change management process.

Example Methods: FRACAS, reliability estimation.

g. Demonstrate Acceptance

Acceptance (qualification, certification, customer demonstration) testing/demonstration is conducted to ensure reliability case evidence is sufficient to meet customer and/or regulatory requirements.

Example Methods: Customer/certification authority system acceptance test suites; reliability case evidence review.

h. Sustain Reliability Throughout In-Service

Sustainment of an acceptable level of software reliability during in-service includes continued attention to reliability data collection and sustainment of the plan/case framework. Collect and analyze failure data during operational use to determine fault categories, sources of failure, and whether operational reliability claims are being met. Sustain/improve the reliability as software is modified during support. Ensure the support concept includes updates as necessary to the software reliability plan/case and associated reliability activities.

Example Methods: FRACAS, customer/certification authority system acceptance test suites; reliability case evidence review; repeat activities as necessary from the development phase.

4.3 Roles and Responsibilities

The roles and responsibilities for system/software reliability are illustrated in Figure 2. The Customer, Supplier, and Certification Authority have special relationships that will vary from project to project based on the application area, criticality of the system, and the formality of the engineering process.

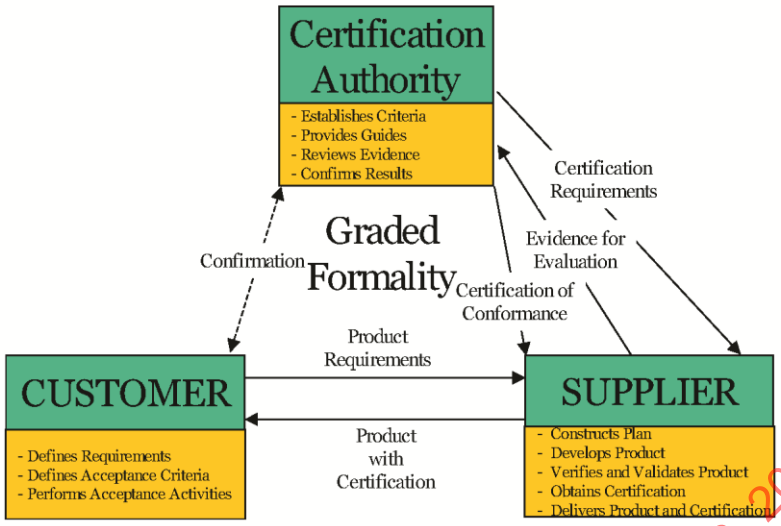

FIGURE 2—CUSTOMER-SUPPLIER-CERTIFICATION RELATIONSHIP

4.3.1 CUSTOMER

The customer represents an organization that procures a system, software product or software service from a supplier. The customer could be one or more of the following: buyer, owner, user, and/or purchaser. It is the responsibility of the customer to establish their requirements, work with the supplier to ensure their requirements are correctly interpreted, and provide appropriate review and approval of the reliability evidence that the delivered product satisfies the specified requirements.

In some cases where regulatory or certification authorities must approve use of the system, the customer will ensure the specified requirements include appropriate regulatory/certification concerns. In addition, the customer must ensure that required supplier interactions, demonstrations, and any other communications with such authorities is conducted as required. Confirmation of certification requirements is coordinated among the customer, supplier, and certification authority.

4.3.2 SUPPLIER/CONTRACTOR

The supplier represents an organization that enters into a contract with the customer for the supply of a system, software product or software service under the terms of the contract. The term "supplier" is synonymous with contractor, producer, seller, or vendor. The customer may designate a part of its organization as a supplier. The supplier may have many separate entities such as the contract developer, vendor that supplies OTS parts, and so forth.

4.3.3 CERTIFICATION/ACCEPTANCE AUTHORITY

The Certification Authority/Regulator is the organization or person responsible within the state or country concerned with the certification of compliance with the requirements. Certification is the legal recognition by the certification authority that a product, service, organization or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval or other documents as required by national laws and procedures.

A software product may be certified as a separate item, a component part of a hardware component certification, or a software component part of a complete system depending on the application. The certification authority may use independent assessment personnel, or designees within the supplier organization that have been approved for the independent assessment role. Other combinations are possible depending on the country, customer organization, application, and critical nature of the software.

5. Task Activities

The purpose of this section is to outline specific task activities that will support achievement of a reliable software product within the context of the Software Reliability Program Standard [JA1002] and the system reliability standard and guidelines presented in references [JA1000] and JA1000-1]. Software reliability cannot be improvised. It must be addressed consistently and systematically, and as an integrated part of system reliability. Reliability must be addressed throughout the complete life cycle from concept through development, delivery and operational use, in a way that allows demonstration to the customer that reliability requirements are being met. Addressing reliability early in the life cycle can reduce the possibility of costly product redesign. The broad approach to ensure software reliability, is illustrated in Figure 3, and based on three simple principles:

a. determine customer software reliability requirements;

b. meet customer software reliability requirements; and

c. demonstrate that the software reliability requirements have been satisfied.

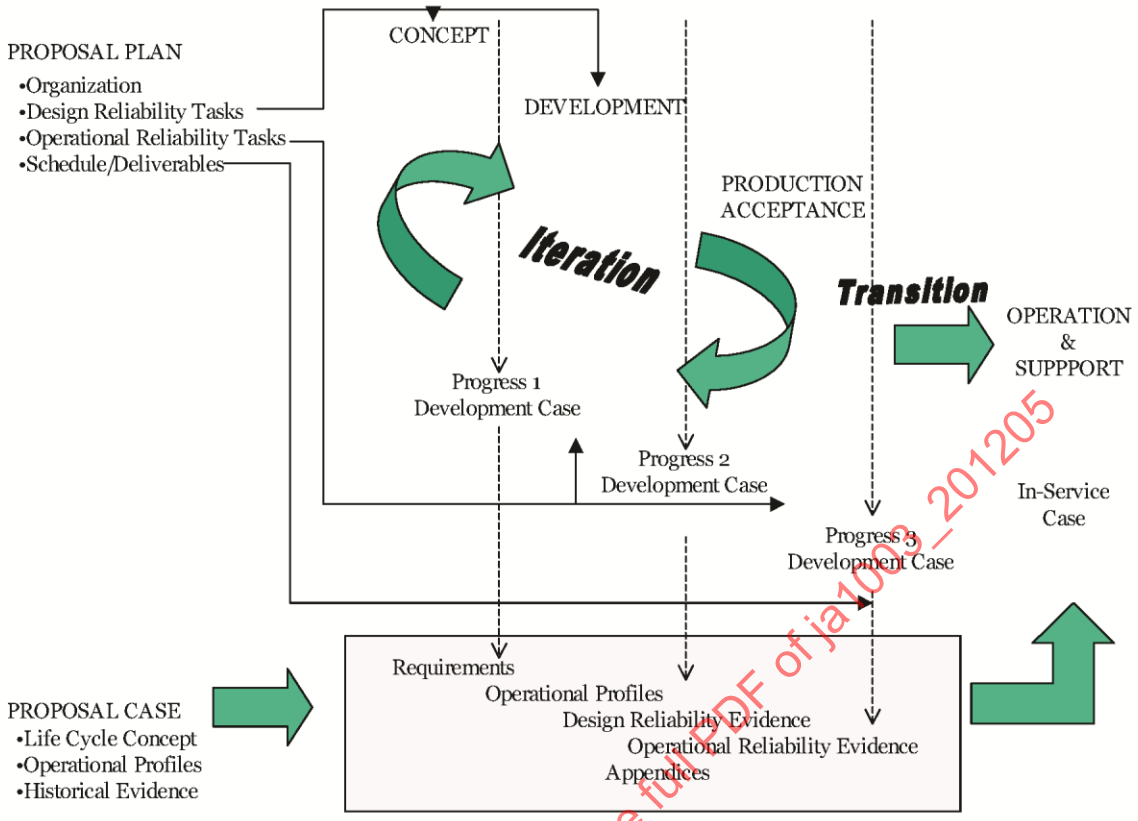

FIGURE 3—SOFTWARE RELIABILITY PROGRAM PLAN-CASE FRAMEWORK

A software reliability program defines the activities necessary to negotiate with the customer what the software reliability requirements are, identify and conduct analyses that ensure customer reliability requirements are met, and provide demonstrated evidence that the customer reliability requirements have been achieved. The identified activities along with measures of acceptance constitute the software reliability Plan. These activities include life cycle practices, methods, and processes that define how the software is designed to meet reliability as well as how the software is operationally tested to ensure that requisite reliability has been attained. The results gathered from the analysis activities constitute the software reliability Case.

A standard Plan and Case mechanism is the recommended approach to documenting a software reliability program [JA1002]. This Plan-Case structure should be implemented as early as possible in the life cycle and continued throughout the In-service phase. Suggested templates for the Plan and Case content are illustrated in Appendix B and described in more detail later in this section.

5.1 Reliability Analysis Tasks

Software reliability analysis tasks are defined within the context of achieving overall system reliability. The context for system reliability and the software reliability tasks presented in the following subsections is consistent with references [JA1000] and [JA1000-1]. The reliability analysis tasks are conducted at various points throughout the product life cycle phases depending on the iteration of the product design, changes to requirements, and the support concept.

A software reliability program should include management tasks such as: life cycle standard process standard (C.4.3) to define the overall software/quality engineering activities; requirements management (C.4.5) to elicit, analyze, document, and manage changes to the customer requirements and any derived requirements; risk management (C.4.6) to balance trade-offs among reliability cost, schedule, and performance; process assessment (C.4.4) to determine at milestone points whether processes, methods and techniques are effective and whether reliability goals appear to be achievable/achieved; configuration management (C.4.1) to control analysis results and reliability case evidence throughout the life cycle; and FRACAS (C.4.2) to ensure appropriate reliability data collection, analysis, and root cause can be conducted.

5.1.1 DETERMINE CUSTOMER REQUIREMENTS

Determining customer requirements requires customer-supplier negotiations, identification of the operational use profiles, definition of precisely what reliability means in terms of failures and criticality levels, and defining in-service conditions of use. Determining customer requirements analysis tasks will generally occur during the Concept and Feasibility phase with iteration across all other life cycle phases as necessary to update and manage changes to the requirements.

5.1.1.1 Establish Customer-Supplier Dialogue

Customer-supplier dialogue can be established through several activities that are a normal part of system requirements elicitation, analysis, and documentation. If certification and approval by a certification authority are also required, then the dialogue should also involve the certification authority. The key aspect of interest for this task is to determine the Goals for software reliability in the context of system reliability goals, what specific software reliability questions are to be answered (e.g., what does “failure” mean?), and precisely what type of evidence would be suitable for answering those questions.

Typical methods/techniques that might be used include: quality function deployment (C.1.11), goal-question-metric (C.1.6), formal scenario analysis (C.1.5), and various types of customer surveys and interactions intended to clarify information provided by the customer and/or the supplier.

5.1.1.2 Identify Operational Conditions of Use

One of the key activities of a reliability program is to determine the profiles of the product's operational use. For software this involves determining the complete set of operations (major system logical tasks) with their probabilities of occurrence. Typical methods/techniques that might be used include: formal scenario analysis (C.1.5), reliability bench marking (C.3.8) in combination with customer surveys and interactions with operational personnel.

5.1.1.3 Define In-Service Conditions of Support

It is important to understand the conditions under which the software is to be supported and how the support (e.g., field loading of software) might affect the approach for establishing a software reliability program. The references [JA1004], [JA1005], and [JA1006] provide a comprehensive approach to addressing software support and supportability. In addition, typical methods/techniques that might be used for software reliability aspects of support include: quality function deployment (C.1.11), formal scenario analysis (C.1.5), reliability bench marking (C.3.8), and various types of using/support organization surveys and interactions intended to clarify information provided by the customer and/or the supplier.

5.1.1.4 Establish Metrics: Goals, Assumptions and Claims, and Expected Evidence

Quantitative and qualitative measures of success need to be established as early as possible. These measures may vary over time as the requirements stabilize and the design is realized. Quantitative software reliability measures such as expected failure rate at delivery must be allocated from system reliability measures. Other quantitative software reliability measures such as defect removal efficiency or fault density must be derived from an understanding of the target failure rate as well as historical data. Qualitative measures need to be defined in terms of an iteration of demonstrated attention to lack of defects in the software specifications, architectural design, and code implementation. Evidence of quality management and engineering practices may be included in the qualitative measures. The negotiation and selection of these measures must be done very judiciously. Every measure must have a definite purpose in the decision-making processes otherwise the measures my become counterproductive. The evidence and associated measures will depend on the type of system. For example, possible failure measures of interest are illustrated in Table 5.

TABLE 5—EXAMPLE RELIABILITY FAILURE MEASURES BY APPLICATION TYPE

Metric	Example	Applicability
Failure On Demand	0.001 ➔ 1 out of 1,000 service requests results in a failure	Transaction systems e.g., ATM or data communication switch
Rate of Failure Occurrence	0.002 ➔ 2 failures are likely in 1,000 operational time units	Periodically operating systems e.g., CAD system, flight simulator, office automation software
Mean Time to Assist Rate of Assists	500 ➔ the time between assists is 500 time units 0.002 ➔ 2 assists are likely in 1000 operational time units	Production/manufacturing systems e.g., Microelectronics control software
Mean Time to Failure	1,800 ➔ the time between failures is 1,800 time units	Continuously operating systems e.g., Nuclear reactor control system, life support system
Operational Availability	0.999 ➔ the software is available for 999 out of 1,000 time units.	Continuously operating systems e.g., Nuclear reactor control system, life support system

Typical methods/techniques that might be used include: quality function deployment (C.1.11), goal-question-metric (C.1.6), formal scenario analysis (C.1.5), reliability allocation (C.1.12), reliability block diagrams (C.1.13), six sigma (C.1.16), SRE (C.1.20), statistical analysis (C.1.21), coverage analysis (C.3.3), dynamic test methods (C.3.4), formal in-process reviews (C.3.5), reliability estimation modeling (C.3.9), and process assessment (C.4.4).

5.1.1.5 Develop Plan

Once reliability requirements are reasonably understood through customer-supplier negotiation, the initial plan of reliability analysis tasks can be established. This plan may include tasks already completed or in progress, such as those tasks associated with determining customer requirements. The plan should be a documentation of the agreed-upon claims and evidence along with the selected analysis, design, and verification tasks that will be conducted. The plan should be approved by the customer and certification authority as appropriate. The Software Reliability Plan template illustrated in Appendix B provides an indication of information that might be included in the plan. Potential methods/techniques to support the task activities are outlined in Appendix C. These methods/techniques do not encompass all that exist. There are existing methods/techniques not mentioned in the appendix and new ones being developed that may be more suitable. However, the basic guidelines presented in this document should still provide a valid framework for selecting the most appropriate methods/techniques depending on the application and customer requirements.

5.1.1.6 Document Pre-Development Case Evidence

The collection of software reliability evidence created during the pre-development life cycle activities should be documented as the initial software reliability case. The Software Reliability Case template illustrated in Appendix B is an example format for information that might be included in the case. See 5.2 for more guidelines on the case documentation.

5.1.2 MEET CUSTOMER REQUIREMENTS

Meeting customer requirements requires definition of an appropriate software life cycle model that includes the planned reliability tasks, including interaction with systems engineering; conduct of static and dynamic analysis tasks with results reporting, performance of design, implementation, and test activities; continual assessment and management of reliability risk; and documentation of the development case of evidence. Meeting customer requirements tasks typically are the focus of the development and production life cycle phase, but may be used during the prototyping of design concepts as well as during the operation phase to show requirements are still being met and as part of the support activities to show updated software still satisfies the requirements.

5.1.2.1 Define Life Cycle Model and Interaction with System Engineering

One of the principles of this guideline document is that good software engineering practices embedded within a well-defined life cycle model will provide qualitative and quantitative evidence of improved software reliability. Qualitative evidence can be provided in the form of statistical correlation of specified practices with fewer delivered defects. Quantitative evidence can be provided in the form of reduced failures as observed in test results or in-service FRACAS (C.4.2) records.

The software engineering model should be an integrated part of the overall system engineering approach. System reliability activities will help define the scope of the software reliability program and its integration within the system/software engineering model. Management techniques of particular interest include: tailoring a life cycle model from appropriate life cycle standards (C.4.3); use of requirements management (C.4.5) and risk management (C.4.6) to manage an integrated approach to failure prevention, detection, removal, and mitigation; and process assessment (C.4.4) to identify system engineering areas of improvement. Other techniques that can be leveraged with system engineering management might include: common cause failure analysis (C.1.2), hazard analysis (C.1.7), probabilistic methods (C.1.9), probabilistic testing (C.3.4), reliability block diagrams (C.1.13), reliability prediction modeling (C.1.14), six sigma (C.1.16), software FMECA (C.1.18), software FTA (C.1.19), statistical analysis (C.1.21), reliability allocation (C.1.12), reliability bench marking (C.3.8), and reliability estimation modeling (C.3.9).

5.1.2.2 Design Reliable Solution

It is important to design software with the specific intent to be reliable. This requires considerable study and analysis of architectural options and translation of the options into a detailed design specification. The design of the selected architectural option should make it more efficient to demonstrate that requirements are satisfied or still satisfied if changes are made. Reliable architectural designs target characteristics of simplicity, modularity, and cohesion to facilitate the prevention, detection, removal, and mitigation of errors, faults, and failures of the software. Adopting an architecture and detailed design that includes interface verification methods (static and dynamic) facilitates defect prevention, identification, mitigation, and removal during both development and support activities. Software language selections to support implementation of such architectural design principles is an important consideration and should be done during design in order to enable a seamless transition to implementation, operation, and support. The language decision is a fundamental design decision that will affect production, testing, training, and support. Language implementation features to facilitate reliable solutions will be briefly discussed in 5.1.2.3.

Some of the techniques and methods that facilitate obtaining a reliable architectural design include: formal scenario analysis (C.1.5), reliability prediction modeling (C.1.14), response time, memory, constraint analysis (C.1.15), design by contract (C.2.1), engineering process maturity (C.4.4), fault tolerant design (C.2.2), information hiding (C.2.4), mistake/error proofing (C.2.5), reliability allocation (C.1.12), requirements analysis and traceability (C.4.5), boundary value analysis (C.3.1), cleanroom (C.3.2), peer reviews (C.3.7) and formal in-process reviews (C.3.5). When safety/security-critical requirements are important, techniques and methods that complement reliability assurance include: hazard analysis (C.1.7), Petri nets (C.2.6), software FMECA (C.1.18), software FTA (C.1.19), formal methods/languages (C.2.3), and I4 (C.2.4). Further discussion of safety and security considerations is in 6.2.

Nearly all of the Analysis and Design, and many of the Verification methods/techniques listed in Appendix C are possible selections for this part of a Software Reliability Plan. Many of the methods/techniques overlap or work in combination with other methods/techniques. It is important to tailor the selections to achieve a cost-effective reliability program.

5.1.2.3 Implement Reliable Solution

The process of implementing a design consists of translating the detailed design information into a software code form and then verifying that the implementation satisfies the design, and ultimately the specified requirements. To accomplish this it is necessary to maintain traceability of requirements (C.4.5) throughout the entire software life cycle process. Typical life cycle activities of coding, unit testing, analyzing various coverage measures, and conducting reviews should include attention to reliability-specific methods and techniques. Operational and support procedures should be developed to ensure software is reliably loaded, initialized, initiated and, as appropriate, replaced during field use.

The implementation may vary depending on the target language, operational environment, and executable medium. Reference [XTALK03] contains several articles that describe languages features, language selection, and implementation principles that may assist or hinder a reliable, safe, and secure solution. In particular, for most of the techniques discussed in Appendix C the language selection can facilitate or hinder the use of the technique. In some cases, the language completely or partially implements the technique. Language features such as strong type checking, assertion checks, formal exception handling, modularity constructs, compiler validation, and multiple platform implementations facilitate and/or implement such design techniques as: design by contract (C.2.1), degraded mode/fault tolerant capabilities (C.2.2), formal methods/languages (C.2.3), I4 (C.2.4), mistake proofing (C.2.5), requirements analysis and traceability (C.4.5), and nearly all of the verification techniques (C.3) appropriate for the lower level verification activities. In addition, these implementation language features and a support system of robust automated tools can greatly aid key reliability analysis techniques such as Petri nets (C.2.6), probabilistic methods and risk assessment (C.1.9 and C.4.6), response time/memory/constraint analysis (C.1.15), software FMECA (C.1.18), software FTA (C.1.19), software reliability engineering (C.1.20) and statistical analysis (C.1.21).

If the software is implemented in hardware, such as Application Specific Integrated Circuits (ASICs), then hardware (e.g., hardware description language with commercial tools for model checking and verification of designs, fabrication technology, built-in tests, production and acceptance tests) techniques must be integrated as well to provide evidence of meeting reliable functional, electrical, and environmental requirements.

The bottom line is to implement a software solution that is verified at the lowest levels against the design intent. It is also important that the implementation provides features to assist the analysis, verification, validation, and continued evaluation that demonstrates the software meets its customers' requirements.

5.1.2.4 Verify and Validate Solution is Reliable

Once the software is implemented and lower level verification has been achieved, it is important to verify and validate that the software solution interfaces with a representative operational system environment as required and expected. This involves integration and system testing, use of operational profiles, coverage analysis within the scope of the operational profiles, and specific reviews to ensure any special cases (such as non-reproducible hardware error processing or safety/security functions) are adequately verified. Validation of execution results against experimental data or known historical and/or operational data is an important activity. Software reliability prediction and estimation provide key decision information and facilitate identification of risks and appropriate risk mitigation strategies. All the verification techniques (C.3) are candidates for use, but those of particular importance include: boundary value analysis (C.3.1), coverage analysis (C.3.3), multiple testing techniques (C.3.4), software reliability estimation modeling (C.3.9), and software reliability prediction modeling (C.1.14). These verification techniques provide data for analyses critical to understanding the reliability of the software component as well as the system.

5.1.2.5 Assess and Manage Reliability Risk

The risk management (C.4.6) approach should include methods to assess how well reliability requirements are being achieved throughout the development (and support) phases. Typical methods/techniques such as system/software FMECA (C.1.18) and system/software FTA (C.1.19) establish potential high risk areas. Methods/techniques such as reliability prediction modeling (C.1.14), reliability estimation modeling (C.3.9), and the numerous static and dynamic testing methods provide inputs to the risk assessment. Methods/techniques such as probabilistic methods (C.1.9), probability/reliability-based risk assessment (C.4.6), and statistical analysis (C.1.21) provide confidence measures that support risk assessments. More comprehensive software reliability program techniques such as SRE (C.1.20) in conjunction with the use of formal scenario analysis/operational profiles C.1.5) provide a fundamental process within which most of the above techniques can be integrated to assess and manage reliability risk.

5.1.2.6 Document Development Case Evidence

The collection of software reliability evidence created during the development life cycle activities should be documented and appended to the initial software reliability case defined in 5.1.1.6. The Software Reliability Case template illustrated in Appendix B is an example format for information that might be included in the case. See 5.2 for more guidelines on the case documentation.

5.1.3 DEMONSTRATE CUSTOMER REQUIREMENTS

Demonstrating that customer requirements have been met involves the more formal aspects of qualification of the product and/or processes for customer and/or certification acceptance; establishing process controls to ensure reliability requirements can be sustained; transitioning the software product to the operational environment; training end-users and support staff; ensuring the capability to pursue continuous improvement is established; establishing a sustaining data collection capability to track failures and other pertinent reliability information; and documenting the post-development software reliability case for ultimate delivery to the customer support organization. Demonstrating customer requirements are satisfied is a focus of post-production and delivery for In-Service support, but selected demonstration activities may occur throughout the life cycle.

5.1.3.1 Qualify/Certify the Product and Process

The software reliability case should provide the required evidence that the software product, and the life cycle processes used to produce the product, has met all customer reliability requirements. The complete set of verification and validation evidence, of which the software reliability case is potentially a critical subset, is required for complete product qualification and certification per the certification authority requirements. Evidence and perhaps specific demonstrations that the software product satisfies the customer operational scenarios/profiles within a representative system operational environment is a critical part of the qualification and certification activities.

5.1.3.2 Establish Process Controls

The qualified software product must be controlled through a formal configuration management system (C.4.1). This system includes: (1) complete identification of the software product (including life cycle artifacts such as requirements, design, implementation, test, build); (2) a change management capability for controlling failure information, change requests, and traceability for software product versions; and (3) a transition and migration process for controlling distribution of the software product from the supplier to the customer. All of this would be part of a more comprehensive software support concept in which the software reliability plan and case construct would be sustained as the software product is modified.

5.1.3.3 Transition to Operational Environment

The actual transition of the software product from the supplier to the customer's operational environment (initial version or subsequent updated versions) should done in accordance with a well-defined migration plan. This plan would include the transition of the software reliability plan/case information to the software support agency as well as training on what is required to sustain the software reliability program during field installation, operational testing and use, and logistic functions that are part of the software support concept. Support tasks may be a root cause for failures that should be considered as part of an SFMEA (C.1.18) or SFTA (C.1.19). Support tasks may require their own Formal Scenario Analysis (C.1.5) to identify potential activities that may have a higher risk of affecting the operational use of the software, such as the field loading of the software. See references [JA1004], [JA1005], and [JA1006] for more information on software supportability and support scenarios.

5.1.3.4 Train End-Users, Operations and Support Staff

The human operators, users, and support staff are an important component of a reliable system. These personnel will generally require training on the proper use of the system, including any interfaces with the associated software product(s). The software reliability program should include activities to identify any training requirements, analyze the most effective presentation method, develop appropriate training materials, and as appropriate, train those personnel who might affect the correct operation of the software. This training program should be developed as part of the customer requirements negotiation, implemented concurrently with the software product, improved through prototype interactions with the customer, demonstrated to the customer through official acceptance training activities, and updated/conducted as necessary by the support agency.

5.1.3.5 Pursue Continuous Improvement

The techniques of customer feedback/surveys for lessons learned, quality metrics (C.1.6 and C.1.11), common cause analysis (C.1.2), reliability estimation (C.3.9), reliability prediction (C.1.14), reliability-based risk assessment (C.4.6), and many others provide valuable feedback for root cause analysis (C.3.10) and general continuous improvement. Comprehensive management techniques such as process assessment (C.4.4) and engineering process maturity provide well-defined methods for pursuing continuous improvement. Software reliability case evidence is a primary source for establishing baseline measures and improvement objectives. The software reliability program is well-suited to tracking improvement goals and whether they have been achieved.

5.1.3.6 Establish Data Collection and Reporting

Formal data collection and reporting should be part of a systems approach typically characterized as a Failure Reporting and Corrective Action System (FRACAS), see C.4.2. Suggested data reporting concepts and existing data repositories are briefly discussed as a special consideration in Section 0.

5.1.3.7 Document Post-Development Case Evidence for In-Service Use

The collection of software reliability evidence created during the post-development life cycle activities related to customer acceptance and demonstrated capability should be documented as the post-development software reliability case. This case information should be appended to the software reliability case defined in 5.1.2.6 as the initial In-Service software reliability case and delivered to the customer support organization for sustained use. The Software Reliability Case template illustrated in Appendix B is an example format for information that might be included in the case. See 5.2 for more guidelines on the case documentation.

5.2 Reliability Case Documentation

The software reliability case captures the assumptions, claims, arguments, and evidence necessary to provide a convincing argument that a software product has a specified reliability. This collection of information, referred to as the Software Reliability Case, must be accurate, current, and complete and presented in a convincing manner in order to obtain sign-off by the customer and/or certifying authority. It consists of six components: system context description; software reliability goals, objectives, and requirements; assumptions and claims; evidence and reasoned arguments to support the claims; conclusions / recommendations; and certification records. There are essentially two types of evidence: (1) direct evaluation of the achieved reliability of the software products, and (2) evidence of the general suitability of the software engineering process. See Appendix B for a sample software reliability case file outline and FRACAS (C.4.2) for a discussion of case interface with a data collection and storage capability. This section provides an overview of the software reliability case as illustrated in Figure 4. Additional suggestions for reliability case information specific to life cycle phases is contained in reference [DEFS0042-3].

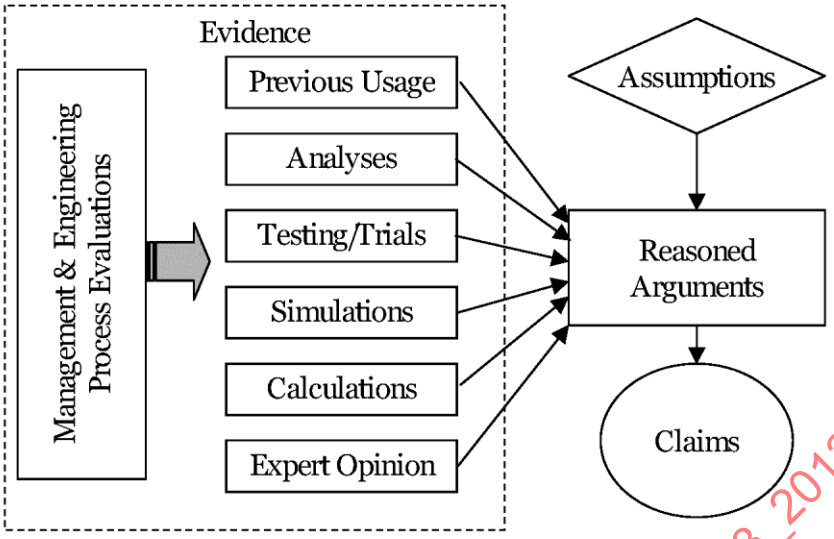

FIGURE 4—RELIABILITY CASE: CLAIMS BASED ON EVIDENCE

5.2.1 SYSTEM CONTEXT DESCRIPTION

The system description should provide a context for understanding the software component(s) in relation to the system. The description should briefly include the: system's physical equipment characteristics; system's physical boundary such as through block diagrams; system's primary role or function and any secondary roles in order to provide a view of its operational criticality – e.g., typical peacetime and wartime mission profiles; system's operating environments; interfaces with any equipment associated with the inputs, outputs, and services the software provides to the system; configuration build of the system/software for which the case evidence applies; personnel skill levels and training that are required to operate and/or maintain the equipment/software; and the maintenance policy/support concepts for each of system's role or mission profiles.

5.2.2 GOALS, OBJECTIVES, AND REQUIREMENTS

The information in the software reliability case file must correlate with the specified software reliability requirements. Hence, the software reliability goals, objectives, and specific requirements should be stated first, for the system and individual partitions, as appropriate. The process by which the reliability requirements were derived and apportioned to software should be described. The relationship between the system and software reliability requirements should be explained. Any regulatory and/or contractual reliability requirements should be highlighted. In addition, the agreed upon validation and certification criteria should be noted. Any risk areas associated with the system/software satisfying the reliability requirements should be identified along with how these risks will be, or have been managed. Based on the risks identified, the maturity of the proposed design solution and the reliability requirements, the strategy adopted to meet the requirements and provide the necessary assurance should be summarized. This strategy is justification for the reliability program activities and the identification of the success criteria for these activities.

5.2.3 ASSUMPTIONS AND CLAIMS

All assumptions, such as citing existing systems or research, and claims made relative to achievement and assessment of the software reliability goals and objectives should be clearly stated and justified. A good record in the past is not sufficient as evidence of good performance to be expected in an environment that is modified even slightly. Any change in the physical, logical, or data environment must be taken into account.

5.2.4 EVIDENCE

Three categories of evidence should be collected iteratively throughout the life cycle and included/referenced in the software reliability case file: process activities, product characteristics, and qualifications of people and resources that demonstrate achievement of software reliability requirements. This information can be summarized in terms of activities conducted, techniques used during the activities, and measures to satisfy claims across process, product, and resources and reliability control areas as illustrated in Table 6:

a. Fault Elimination—Requirements, design, implementation and test prevention/detection/removal/mitigation of defects/faults;

b. Failure Containment—Failure impact limitation; failure isolation and recovery; fault tolerant, degraded mode operations; operational profile coverage testing & analysis; and

c. Failure Rate Estimation—Failure rate prediction from development parameters; FRACAS data collection; failure data analysis & reliability estimation modeling; statistical analysis and probabilistic methods; failure/impact reliability risk assessment

Adequacy of evidence provided for each activity should be evaluated. Professional judgment is required to evaluate the evidence presented. A checklist adapted from [DEFS0042-3] is indicated below as a guide to some of the possible evaluation areas:

a. stated objectives for the activity are clearly defined;

b. activity is systematic and complete;

c. activity been undertaken at a time that will allow influence on the design;

d. usage profile and environmental constraints have been considered for the activity;

e. physical and functional boundaries of the activity have been defined;

f. any assumptions are defined (e.g., inputs from other systems or services), and are realistic and reasonable;

g. justification is provided for the activity method/technique used and is reasonable;

h. description is provided of who (e.g., user, maintainer, designer etc) was consulted during the activity and these personnel and level of consultation is reasonable;

i. activity recommendations are clearly defined and reasonable;

j. documentary evidence indicates that the recommendations have been implemented; and

k. activity results have been progressively updated to reflect the latest design and are being used as an input to design reviews.

TABLE 6—EXAMPLE FORMAT FOR SUMMARY OF SOFTWARE RELIABILITY CASE EVIDENCE

System:_____

Intended Use/Environment:_____

Phase/Date:_____

Reliability Control Measure	Product Evidence/ Safeguards	Process Evidence/ Safeguards	Resource Evidence/ Safeguards
Fault Elimination	- - - - -	- - -	- - -
Failure containment	- - - - -	- - -	- - -
Failure rate estimation	- - - - -	- - -	- - -

5.2.4.1 Process Activities that Demonstrate Achievement of Software Reliability Goals

A description of the selected life cycle model and development method should be provided, including an explanation of how this model and method contribute(d) to the attainment and assessment of reliability goals throughout the life cycle phases. Specific life cycle activities that were used to assess software reliability should be called out, such as performing iterative risk analyses or using static analysis techniques. An assessment should be made of the:

- software reliability design analysis;
- software reliability code analysis;
- software reliability change analysis; and
- effectiveness of validation and verification activities.

Suspected or confirmed reliability problems should be documented, along with the current status of their resolution. Results from analyzing and interpreting process metrics should also be discussed.

5.2.4.2 Product Characteristics that Demonstrate Achievement of Software Reliability Goals

A description of the design features that contribute to enhanced reliability should be provided, such as: design by contract (C.2.1), I4 (C.2.4), and system/software fault tolerance design (C.2.2). This description should explain how the likelihood of common cause failures (C.1.2) has been eliminated or reduced, through use of techniques such as system/software FMECA (C.1.18) and system/software FTA (C.1.19). In addition, a discussion of whether the product: (1) operates in a demand-mode or continuous-mode environment; (2) was designed to fail safe or fail operational; and (3) contains any monitoring and/or error detection and correction features should be included. The results of static and dynamic test analyses should be recorded, along with an analysis of the effectiveness of the reliability control measures. Results from analyzing and interpreting product metrics should also be discussed. A well-planned dynamic test program is an essential part of providing confidence in the reliability of software. Dynamic testing evidence (C.3.4) should be provided along with coverage analysis (C.3.3) and evidence that requirements have been traced (C.4.5) to design, implementation, and dynamic tests.

5.2.4.3 Qualifications of People and Resources that Demonstrate Achievement of Software Reliability Goals

An explanation of why the education, experience, and certification of the professional staff are appropriate for a project of this reliability level should be provided. Likewise, a justification of why the hardware and software platforms, including automated tools, are appropriate for this project should be provided. Results from analyzing and interpreting people/resource metrics should also be discussed.

5.2.5 CONCLUSIONS/RECOMMENDATIONS

The conclusions/recommendations should summarize the key information presented in case evidence accumulated to date, such as which software reliability requirements have/have not been met and with what level of confidence. In interim issues, it should recommend whether the project should proceed to its next milestone, or what further work is required to enable the project to progress. In addition, it should recommend what activities should be conducted in the future in order to generate the necessary assurance that the reliability requirements will be satisfied and what residual risks/limitations may constrain the results. For operational use, recommendations should be provided regarding the software qualification/certification status.

5.2.6 CERTIFICATION RECORDS

An accurate and complete chronological history of all certification-specific information and activities should be included or referenced in the software reliability case file. The format and precise content is typically dependent on the certification authority. An example of the type of Federal Aviation Administration (FAA) certification authority interactions and a fragment of potential certification information is provided in Appendix E.

6. Special Considerations

A software reliability program may involve a variety of special considerations, some of which are discussed in this section.

6.1 Tailoring the Software Reliability Program

A layered use of standards and guidelines with appropriate tailoring of activities and techniques is the most effective approach to establishing a software reliability program. Within each of the layers, such as illustrated in Figure 5, there are opportunities to tailor the activities to best suit the application. This tailored set of activities is negotiated as part of the initial negotiation with the customer to determine system/software reliability requirements.

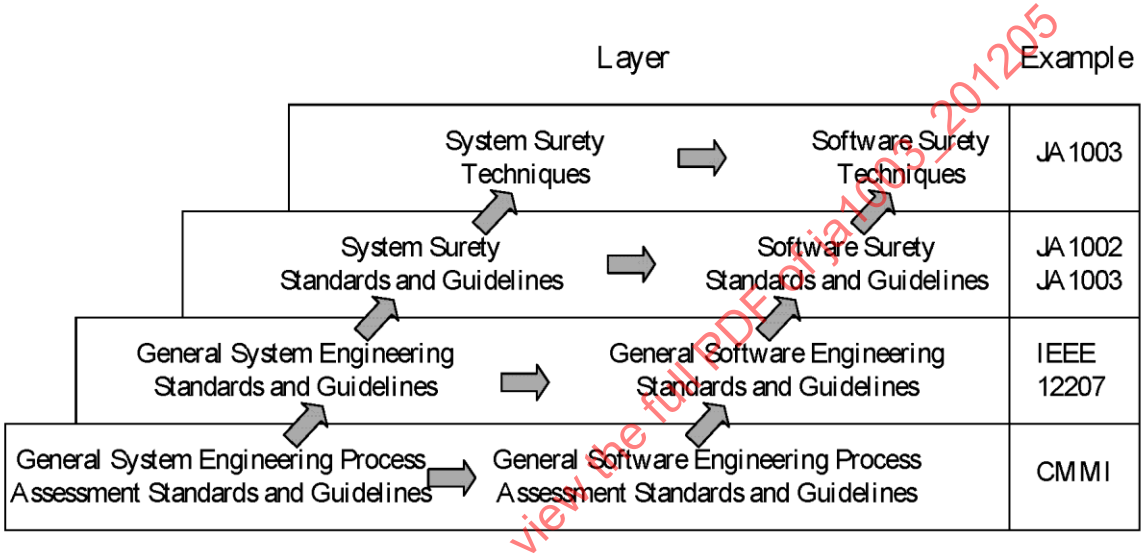

FIGURE 5—LAYERED APPROACH TO SOFTWARE RELIABILITY TAILORING

The tailoring of a software reliability program for a given project will be the focus of implementing the "Determine Customer Requirements" reliability analysis task. The examples in Appendix D and Appendix E provide some instances of tailoring the analysis tasks. Some specific tailoring considerations are discussed in the following subsections. Special tailoring considerations for application variations and complexity, criticality levels, technique selection, safety and security, Off-The-Shelf software, and data collection and reporting are briefly discussed in 6.2, 6.3, and 6.4.

6.1.1 APPLICATION VARIATIONS AND COMPLEXITY

There are many different applications for which the drivers for tailoring will vary. Such applications include military, aerospace, motor vehicle, railway, medical equipment, pharmaceutical, and nuclear industry. Regulations (requirements specific to the domain) are typically developed and managed by an application domain organization. The application regulations will drive the specific system/software engineering processes, verification and validation activities, and in general the reliability, safety, and security techniques that provide the required assurance. Within the context of the application domain regulations, the required integrity level and complexity of the system/software will determine which reliability techniques will be most appropriate to provide the required level of assurance.

Military systems will be driven by the military acquisition system procurement regulations, responsible armed forces regulations, and the subsequent supplier contractual mechanism. Available funding will be driven by government budget reviews and military allocations. The United States Department of Defense military acquisition process also varies from other country's (e.g., United Kingdom Ministry of Defence, Canadian National Defense) military acquisition. Some military/federal regulations will apply as well as industry-specific standards and guidelines. The acquisition system procurement process must require reliability/safety/security plan and case information to be generated as a Key Performance Parameter if it is to be assured. In addition, the requirement should be for acquisition customer-supplier negotiation, analysis, and demonstrated assurance evidence throughout the acquisition and support life cycle. For system acquisition this is directly supported by the system reliability program standard/guideline, [JA1000] and [JA1000-1]. For the software acquisition this is directly supported by the software reliability program standard/guideline, [JA1002] and this JA1003 guide.

Industry systems will be driven by the full range of customer priorities and the enterprise market requirements for time-to-market and profit. An additional driver is the potential for litigation if the system/software product reliability/warranty does not meet expectations. Industry applications such as aerospace, railway, medical, and nuclear energy have significant safety and security concerns. Legal liability can fall on the manufacturer, supplier, distributors, or certifier of products. Such systems/software may have a certification authority provide a required oversight function:

1. Federal Aviation Administration for commercial airlines;
2. Nuclear Regulatory Commission for the nuclear power industry;
3. Her Majesty's Railway Inspectorate for the United Kingdom railway systems; and
4. Food and Drug Administration for pharmaceutical equipment and processes.

Each of these certification authorities will have source regulations, evaluation and assessment procedures, and best practice recommendations that will need to be integrated into the system/software reliability plan/case framework. Associated software will need to apply techniques such as hazard analysis (C.1.7), software FMECA (C.1.18), software FTA (C.1.19), I4 principles (C.2.4), and perhaps even formal methods and languages (C.2.3) to provide adequate assurance that the systems are safe and reliable. When a system/software has significant reliability, safety, or security concerns independent evaluations, assessments, and audits may be part of the required oversight activities. A partial example of establishing a software reliability program for an aerospace application within the requirements of the Federal Aviation Administration is illustrated in Appendix E.

A system/software product may be very complex. This complexity can originate in the nature of the problem being solved by the application, number of functions being provided, number of components with which interfaces are required, or number of applications for which the product is intended to be used. In turn, this complexity manifests itself in the number of system/software requirements, architecture design structure, implemented product decision paths, and level of static and dynamic testing required. A well-planned and executed software reliability program should be tailored to reduce these complexities.

6.1.2 CRITICALITY LEVELS

There are many different concepts for criticality levels related to system/software reliability. The term itself may be "severity", "integrity", "risk", or just "criticality" level. In almost all cases the criticality level is a qualitative ordered scale. The usual association with the criticality level is that the rigor of the reliability evidence required increases with the criticality levels. Specific system/software engineering and reliability/safety-specific activities are defined so there is acceptable assurance the desired criticality level will be attained. Although there are standards and guidelines (e.g., [DO178B]) that specify which activities are to be conducted to satisfy the desired criticality level, such activities are still part of a tailoring process involving the customer and supplier, and as appropriate the certification authority. Thus, the specific task to "Determine Customer Requirements" would involve the specific definition of what failures are of concern, the criticality levels associated with those failures, and what specific reliability tasks must be conducted to provide the required confidence that such failures will not occur.

For example, criticality levels could be based on: level of financial cost impact to the customer, level of functional impact to the customer; reliability/failure rate level; or risk level as a qualitative combination of frequency of failure and impact of failure for each failure category. What is certain is that additional reliability concerns of safety and/or security will increase the level of rigor required by the reliability program. The customer/supplier negotiations are critical to establishing the desired criticality levels. A software reliability criticality level matrix with example software applications is illustrated in Table 7.

TABLE 7—EXAMPLE SOFTWARE RELIABILITY CRITICALITY MATRIX

Catastrophic				Nuclear Weapon/ Safety of Flight
Hazardous/Severe			Nuclear Energy	
Major		Medical Radiation Treatment		
Minor	Information Management			
Hazard Level	10^{-1}	10^{-3}	10^{-5}	10^{-6}
Failure Rate				

Table 7 is only meant to be illustrative since within the indicated applications, the desired failure rate and hazard level combination (risk level) may be different than indicated.

At the "upper" levels of criticality there is the problem of providing sufficient quantitative evidence of very high (e.g., less than 10^{-6} failure rate) or even ultra high (e.g., less than 10^{-9} failure rate) reliability systems. For a given confidence interval, the uncertainty in the reliability estimate determined from testing depends on the number of samples gathered. The more samples, the smaller the uncertainty in the estimate. As an example, for a 95% confidence interval and 20 samples, the reliability estimate would range approximately from 0.70 to 1.60 of the measured average value. However, for 200 samples, the range would be approximately 0.90 to 1.15 of the average.

If the reliability requirement is high, say, no more than 1 failure in 100,000 hours of operation, it could take more than 11.4 years of continuous testing to gather a single data point. A minimum of 20 samples is required to achieve the 95% confidence degree of certainty. Hence somewhere in the order of 228 years of testing would be required. However, if large scale parallel independent testing on multiple units is possible, or the software execution hours are significantly less than the system operational hours, then it may be feasible to demonstrate high reliability through testing. In addition, it may be that only a small portion of the software is required to have high reliability, and can be so demonstrated using these accelerated testing techniques. See the Figure 6 below for an illustration of determining the relationship among defect sample size, reliability multiplier, and confidence limits.

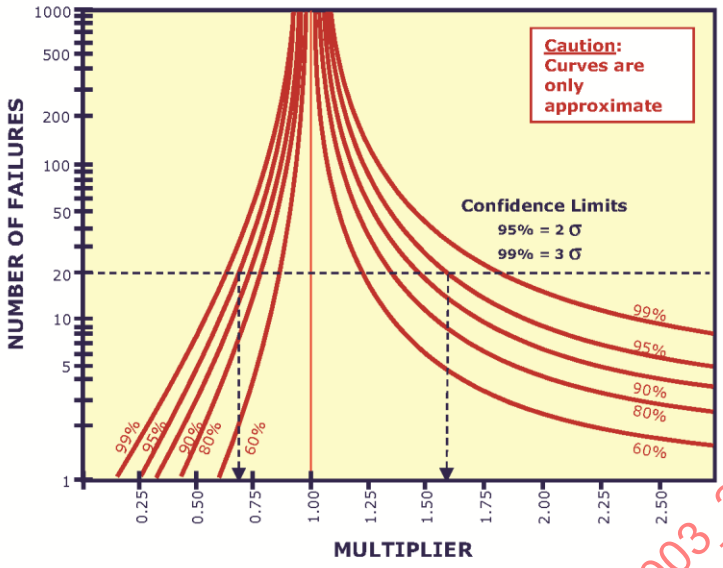

FIGURE 6—RELIABILITY CONFIDENCE LIMITS

6.1.3 TECHNIQUE SELECTION

One of the problems that will need to be resolved by an effective software reliability program is which techniques to select from the many that exist. There are many techniques illustrated in Appendix C, and other existing techniques, and more techniques that will exist in the future. Selecting which techniques will be best suited for a specific software reliability program will be a key activity of the tailoring approach.

There are many decision-theory selection methods to assist in determining which technique(s) from a sample of alternative techniques is more preferable to use. One example is the Pugh selection method (C.1.10). As an example, such a method might be used in combination with selection criteria such as defined in [AIAAR013] or [IEEE1413], for evaluation of alternative software reliability models. In [AIAAR013] the following model evaluation criteria are suggested:

1. model predictive validity for accuracy, trend, bias, and noise;
2. ease of measuring model parameters, including cost and schedule impact;
3. quality of assumptions in comparison with the real world as well as adaptability to special conditions and environments;
4. capability to estimate useful quantities needed by software project personnel;
5. applicability to program evolution and changes in the test and operational environment;
6. simplicity of understanding the concept, data collection, program implementation, and validation; and
7. insensitivity to small differences in input data and parameters without losing responsiveness to significant differences.

Given a set of alternative software reliability models, perhaps as recommended in [AIAAR013], the question to be answered is which one or perhaps hybrid combination of model alternatives would be best for a specific software reliability program? The Pugh selection method (C.1.10) is one possible selection approach that could result in greater insight into the software reliability requirements and solutions. Such results might include a quick review of the many software reliability models and additional hybrid alternatives, a better understanding of the models, a more objective selection process, and increased system/software teamwork and communication.

Selection techniques such as the Pugh method using selection criteria such as in [AIAAR013] and/or [IEEE1413] are valuable in all life cycle phases. These techniques facilitate determining customer software reliability requirements and establishing an effective set of software reliability activities that will provide evidence that the customer requirements have been and are continuing to be met.

6.1.4 DOCUMENTATION

Tailoring the documentation to be an effective information resource used throughout the software life cycle is a challenging task. An effective software reliability program will need to have a well-defined approach to documentation. This approach should state what information is to be captured as well as its form and format.

The basic requirement in [JA1002] is to document the customer requirements, the activities to show the requirements will be met, and the activity evidence and demonstrations that the requirements have been met. The framework recommended for capturing this information is a software reliability plan and software reliability case. The resulting plan/case documentation may be represented in many different information and media formats. There is no "one" way to do it.

For example, the software reliability plan information may be a separate appendix or integrated within a system reliability plan. The plan may be a physical document or an electronic set of information. Alternatively, all system/software reliability plan information may be integrated within an overall system and/or logistics plan, perhaps hyperlinked electronically from an overall system documentation architecture. Similarly, the software reliability case information may be part of a system reliability characterization report that summarizes the case evidence, is presented as part of a web-based information system, and references detailed information contained within a system reliability analysis data base. The system reliability analysis data base may be linked to commercial and industry-specific data related to all the system components, including the software components.

When there are significant safety, security or other surety-related concerns, there will be specific documentation information that has reliability dependencies as well. This documentation should be an integrated part of the specialty engineering and system engineering documentation architecture rather than isolated separate entities. However, there should be filter mechanisms that allow for the extraction of "safety-specific" or "security-specific" or more comprehensive "reliability-specific" information. And, this information should be able to be presented in a variety of documentation formats depending on its use.

One significant software reliability program activity is to define the documentation architecture as related to the selected software reliability activities and the results of those activities. This documentation architecture activity should be an on-going part of the software reliability program throughout the software life cycle. As a minimum there must be software reliability plan information and software reliability case information, however it is physically or electronically represented, stored, or integrated with other system information.

6.2 Safety and Security Considerations

The standards and literature seem to make it clear that, safety, security, and reliability are "different" attributes of a system, at least in the primary focus of these attributes. Software safety is concerned with features and procedures that ensure a software product performs predictably under normal and abnormal conditions. Thereby, the likelihood is minimized of an undesirable event occurring, consequences of any unplanned event are controlled and contained, and accidental injury, death, destruction of property and/or damage to the environment is prevented, whether intentional or unintentional. Software security is concerned with features and procedures that ensure a software product satisfies its requirements for timely access to authenticated services and for protection from denial of authenticated services. On the other hand, the “standard” interpretation for software reliability is concerned with the ability to produce accurate and consistent results repeatedly under low, normal, and peak loading conditions in the intended operational environment. The focus is on operational profiles in the intended operational environment.

However, since software reliability is the probability of failure-free operation of a software program for a specified time under specified conditions, the definition of "failure" will determine the scope of reliability. Unless safety/security requirements are not part of the requirements specification, it does not seem to be possible to have a reliable system that is not safe and/or secure. Whereas, it does seem possible to have a safe and/or secure system that is not reliable. It is clear that if safety and/or security requirements are part of the software specification, then any failure to satisfy those requirements will be interpreted as a reduction in the reliability of the software. Thus, it would seem logical to create an interpretation where the reliability level would be dependent upon whether there were any safety and/or security requirements.

The combination of reliability, safety, and security requirements is sometimes defined as part of a term used in the nuclear industry, "surety". Another term for this combination is “dependability”, as described in reference [LITTLEWD00].

In this guideline document, it is simply acknowledged that safety and security requirements require task activities specifically targeted to providing evidence that those requirements are satisfied in addition to the normal layering of general software engineering requirements and the more usual reliability requirements. These safety/security task activities can either be included within a software reliability plan/case framework or, if significant enough to a certification effort, as separate software safety plan/case and software security plan/case documentation. Also, such information may be integrated at the system level. Hopefully the software design architecture facilitates a focused analysis of a small subset of the software product for either safety or security concerns. The safety and security disciplines provide robust methods that are generally beyond the scope of this document. These crucial subjects deserve much more in-depth treatment than can be described in this document. However, many of the techniques listed in Appendix C could be tailored to support safety and security assurance. Some of the important considerations are introduced in this Section.

Techniques that are more typically applied to high consequence software specifically to address safety and/or security concerns include: I4 (C.2.4); hazard analysis (C.1.7); sneak circuit analysis (C.1.17); software FMECA (C.1.18); software FTA (C.1.19); independent vulnerability analysis (C.2.4); Petri nets (C.3); formal methods/languages (C.2.3); and very high system/software capability maturity (C.4.4).

The themes illustrated in Table 8 have been derived from [HERM99] and augmented somewhat to include the overall aspects of "surety" - encompassing reliability, safety, and security. The themes provide guidance for integration and tailoring considerations.

TABLE 8—MAJOR SURETY THEMES

Surety Principle	Statement
Integration	Software surety is a component of system surety.
Coverage	High integrity, high consequence, mission critical systems need to satisfy all surety requirements. Everyday examples should not be overlooked when classifying systems as surety-critical. The surety principles of independence, isolation, inoperability, and incompatibility apply to all systems.
Specialty Engineering	Special engineering activities are necessary in addition to a well-defined software engineering process to produce safe, secure, and reliable software.
Life Cycle Process	To achieve software surety, certain policy, planning, design, analysis, and verification activities must take place.
Measurement	The achievement of software surety should be measured throughout the lifecycle by a combination of product, process, and people/resource metrics, both quantitative and qualitative.
Resource Skills	Software surety is an engineering specialty that requires specialized knowledge, skills, and experience.
Layered Standards	A layered approach to standards and guidelines is the most effective way to achieve both software and system surety.

6.2.1 SAFETY CONSIDERATIONS

Several standards and guidelines such as [DEFS0055], [IEC61508], and [IEC61511-1] provide guidance for addressing safety considerations, and in particular address software safety. The reference [HERRM99] contains summary descriptions and comparisons of a wide variety of software safety and reliability standards and guidelines including the SAE standards. Publications such as [LEVESON95] and [FALLA96] provide descriptions of best practices along with application limitations.

In [LEVESON95] there is a complete life cycle emphasis on what additional software safety management and technical activities are needed to provide adequate safety assurance in high consequence systems. Some of the key areas which require safety-specific activities include:

1. Managing Safety: The role of management, setting policy, communication channels, setting up a system safety organization, place in the organizational structure, documentation;

2. Integrating the System and Software Safety Process: General tasks allocated to software;

3. Conducting System Hazard Analysis: What it is, how to do it with software components, types of models, types of analysis, current models and techniques, limitations, evaluations;

4. Conducting Software Hazard Analysis and Requirements Analysis: Extension of system hazard analysis;

5. Designing for safety;

6. Design of the human-machine interface;

7. Verification of Safety: Testing, software fault tree analysis; and

8. Validation of Safety: Use of experimental data in a system context, quantification of uncertainty.

In [FALLA96], a safety policy model is used to specify implementation independent safety objectives. Policies or objectives must be specified, verified and validated in a demonstrable manner throughout development, and they form part of a broader safety requirements document. As described in [FALLA96], the development method for a safety policy model consists of six stages that can easily be grouped into a safety plan and safety case.

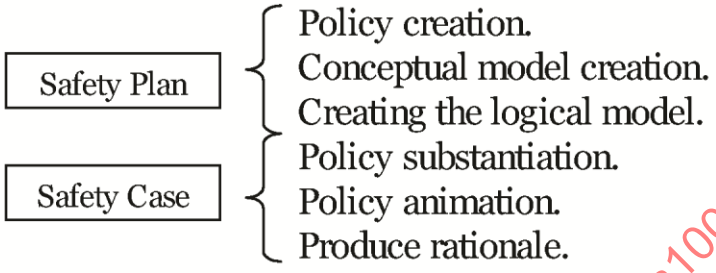

FIGURE 7—SAFETY PLAN, CASE, AND POLICY MODEL

Safety objectives relate to the safe behavior of the system. If these objectives are always satisfied during the system operation then the system will always be in a safe state. The concept is that these behavior objectives can be defined, verified and validated during requirements phases, and verified and demonstrated during design and subsequent phases up to delivery. The role of these objectives and the ensuing safety policy models is to provide key technical criteria for all areas of system development, encompassing the activities involved with design, safety, assessment and management which can then be used as a part of (or be referenced from) a safety case document.

The safety case document could be an integrated part of the reliability case document in that meeting safety objectives and requirements within specified environments is part of a comprehensive reliability program. Or, depending upon the focus of the regulatory certification effort, the reliability case could be presented as part of the safety case. Although the safety policy model may only apply to an isolated part of the software product that specifically controls the safety functions, the concepts could be applied to the more general system/software reliability program.

The SAE Aerospace Information Report 5022 [AIR5022] describes several of the commonly performed Reliability and Safety (R&S) analysis tasks, with emphasis on their inter-relationships and common data elements. That document also describes how the R&S process can be integrated, reducing duplicate work effort and providing more accurate, comprehensive, and standardized analysis results. Several specific reliability and safety tasks are performed on a subsystem of an example product to illustrate how this integration can be accomplished. These tasks are oriented to a system level, but illustrate how the R&S tasks for any component, including the software component, can be leveraged. An example system/software safety life cycle process is illustrated in Figure 8.

In [SSSHDBK99] there is a good list of techniques and methods in Sections 3 and 4 and Appendices C, D, and E. Also, some examples of lesson learned experiences are in Appendix F. [DEFS0055] has a full set of techniques/methods in the following areas:

1. Safety Management: plan, case, analysis, records, reviews, audits

2. Roles and Responsibilities: project, design, V&V, safety auditor, safety engineer

3. Planning Process: QA, documentation, development, risk management, V&V, configuration management, safety methods, design/coding practices, language, tools, previously developed/commercial/diverse software; supportability considerations

4. Development Process: development principles, requirement, specification/design/coding, testing/ integration

5. Certification: qualification, acceptance, training

6. In-Service/Support: change management, safety re-certification

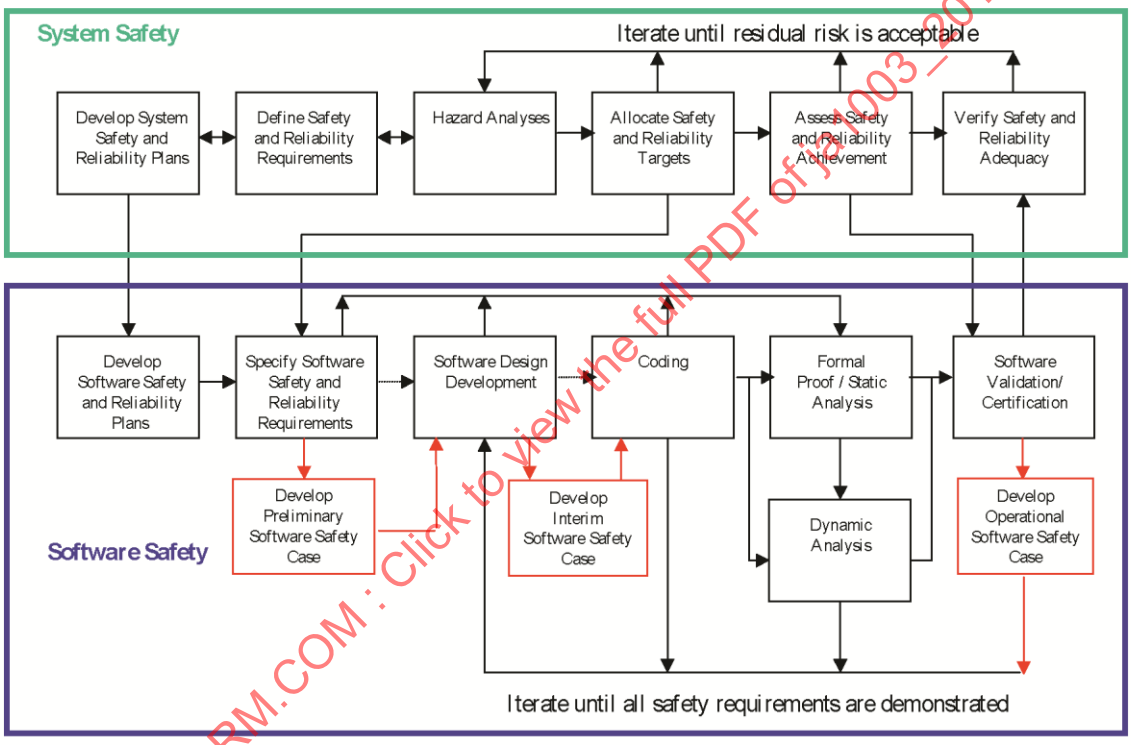

FIGURE 8—EXAMPLE SYSTEM/SOFTWARE SAFETY LIFE CYCLE PROCESS

6.2.2 SECURITY CONSIDERATIONS

Software security includes features and procedures that ensure a software product satisfies its requirements for timely access to authenticated services and for protection from denial of authenticated services. Securing information and systems against the full spectrum of threats requires the use of multiple, overlapping protection layers addressing the people, technology, and operational aspects of information technology. Thus, security can only be achieved by taking a systems approach, which includes features and procedures such as physical security and operational security that are outside the scope of hardware and software.

Software security features and procedures are an important part of nearly any system security implementation. By using multiple, overlapping protection layers, the failure or circumvention of any individual protection layer will not leave the system unprotected. Through user training and awareness, well-crafted policies and procedures, and redundancy of protection mechanisms, layered protections enable effective protection of information technology for the purpose of achieving mission objectives.

The reference [NIST800-27] enumerates a set of information technology security principles that have similar elements as the development method for a safety policy model as presented in [FALLA96] and the surety themes in Table 8. The principles are summarized in Table 9 organized in accordance with the associated surety theme. Most of the principles fall within the "coverage" surety theme. In particular, the security focus on ensuring access control, data integrity, and denial of service involves designing, implementing, and demonstrating that the system/software product follows the principles of Independence, Isolation, Inoperability, and Incompatibility (I4). Security integrity checks (C.2.7) can be used to support access control and data integrity as well as run-time component/data configuration compatibility to prevent indirect creation of backdoors and/or viruses. A security plan/case framework to implement the principles of determining, meeting, and demonstrating customer security requirements can be applied as part of a software security program, or more generally a software reliability/surety program.

TABLE 9—SECURITY PRINCIPLES AND ASSOCIATED SURETY THEME

Surety Theme	Security Principle Statement
Life Cycle Process (Policy)	Establish a sound security policy as the “foundation” for design.
Layered Standards	Implement tailored system security measures to meet organizational security goals.
Layered Standards	Where possible, base security on open standards for portability and interoperability.
Resource Skills	Ensure that developers are trained in how to develop secure software.
Integration	Treat security as an integral part of the overall system design.
Measurement	Reduce risk to an acceptable level.
Measurement	Identify potential trade-offs between reducing risk and increased costs and decrease in other aspects of operational effectiveness.
Coverage (I4)	Clearly delineate the physical and logical security boundaries governed by associated security policies.
Coverage (I4)	Assume that external systems are insecure.
Coverage (I4)	Implement layered security (ensure no single point of vulnerability).
Coverage (I4)	Strive for simplicity.
Coverage (I4)	Design and operate an IT system to limit vulnerability and to be resilient in response.
Coverage (I4)	Minimize the system elements to be trusted.
Coverage (I4)	Implement security through a combination of measures distributed physically and logically.

Surety Theme	Security Principle Statement
Coverage (I4)	Limit or contain vulnerabilities.
Coverage (I4)	Formulate security measures to address multiple overlapping information domains.
Coverage (I4)	Isolate public access systems from mission critical resources (e.g., data, processes, etc.).
Coverage (I4)	Use boundary mechanisms to separate computing systems and network infrastructures.
Coverage (I4)	Use common language in developing security requirements.
Coverage (I4)	Design and implement audit mechanisms to detect unauthorized use and to support incident investigations.
Coverage (I4)	Authenticate users and processes to ensure appropriate access control decisions both within and across domains.
Coverage (I4)	Use unique identities to ensure accountability.
Coverage (I4)	Implement least privilege.
Coverage (I4)	Do not implement unnecessary security mechanisms.
Coverage (I4)	Protect information while being processed, in transit, and in storage.
Coverage (I4)	Develop and exercise contingency or disaster recovery procedures to ensure appropriate availability.
Coverage (I4)	Ensure proper security in the shutdown or disposal of a system.
Coverage (I4)	Protect against all likely classes of “attacks.”
Coverage (I4)	Identify and prevent common errors and vulnerabilities.
Specialty Engineering	Provide assurance that the system is, and continues to be, resilient in the face of expected threats.
Specialty Engineering	Design security to allow for regular adoption of new technology, including a secure and logical technology upgrade process.
Specialty Engineering	Strive for operational ease of use.
Specialty Engineering	Consider custom products to achieve adequate security.

6.3 Off-The-Shelf Software and Reuse

Off-The-Shelf (OTS) software includes all software products that are already developed and available, usable either "as is" or with modification. The most prominent of the OTS software are provided by commercial vendors, government repositories, or open source resources. Most OTS software products might be categorized as "Software Of Unknown Pedigree" (SOUP), meaning there is not full and complete access to the source code, documentation and/or development history. OTS software products are used every day in the form of operating systems, database systems, network/communication systems, compilers, module libraries such as for scientific functions, design support tools, and so forth. The reliability concerns depend on the required integrity level of the application system and the dependence on the OTS software. Several source documents provide guidance on the use and reuse of OTS software products, including references [NUREG6421], [NATO96], [NATO97], [JA1005], and [DEFS0055].

OTS software may be provided as one or more of the following categories:

1. Commercial—provided by a commercial supplier and used with no modifications;
2. Commercial—provided by a commercial supplier with customer parameter inputs to tailor the package for application use;
3. Government—provided by a government agency with the same possibilities as items 1 or 2 above;
4. Industry—provided by an industry/internal organization partner (perhaps another organization/department within a company) with the same possibilities as items 1 or 2 above; and
5. Module Extraction—subpart of an OTS software package is extracted for integration into a new application.

The OTS software may be used in a variety of applications such as:

1. Embedded in Hardware—Communication data link, programmable array logic;
2. Embedded as part of a configurable software/hardware system such as a distributed control system;
3. Integrated as components into a larger system: operating system, relational data base management system, graphical users interface (GUI);
4. Stand-alone Products—Payroll, accounting software, 'shrink-wrapped software' (word-processing, spreadsheets, games);
5. Standard tools used to support the software development and maintenance process: compilers and configuration management tools; and
6. Software libraries and component-based software engineering products.

Potential benefits of OTS software include:

1. reduced time-to-market;
2. decreased development costs;
3. possible higher reliability based on wide-spread operational use; and
4. potential for reuse.

There are acquisition policy influences and pressures to provide new and updated system capabilities at much reduced costs and time to market. The policies of industry as well as government emphasize the integration and use of OTS products (hardware and software) whenever possible rather than develop the new capabilities in-house or through contract. There are reliability and supportability concerns and potential benefits for this approach. Some of the OTS software concerns include:

1. Functionality may not meet all requirements or may provide undesirable functionality that affects system/software reliability;
2. Faults and other problems with the software product will tend not to be made apparent by the supplier, hence may affect software reliability;
3. Obsolescence may be a problem; upgrade releases may be frequent and irregular; if upgrades are required to be integrated for continued supplier support, this may have a detrimental effect on reliability and supportability; support for obsolete software products will tend to be very expense, if not impractical;
4. Integration with bespoke (custom) software and hardware may require special OTS software wrappers; these wrappers are not always easy to develop and may not retain reliability and supportability characteristics of the OTS software;
5. Stated conformance to a standard is no guarantee of interoperability;
6. Testing to validate integrated performance may be extensive, particularly if upgrades are too frequent;
7. Specialty engineering activities to ensure reliability, security, and safety requirements may be difficult, and may require external certification (e.g., trusted software certification); repeat of such activities upon OTS software upgrade would be costly;
8. Support activities by the bespoke software developer (where OTS software is an integrated part of a larger system) can be extensive due to embedded OTS updates/upgrades (frequency and/or complexity of upgrade);
9. Installation, acceptance and training for new releases; and
10. Response of the OTS software supplier to correction of defects may not satisfy operational reliability and/or supportability needs.

However, the OTS software product may provide an exceptional cost benefit with better reliability and supportability than a custom equivalent if it is produced by a supplier who deploys exceptional engineering practices, is cognizant of surety concerns and can provide documented surety evidence, and is responsive to the customers needs. This is a very demanding condition for both supplier and customer and typically can not be done cheaply. As an example, nearly all operating systems are provided through commercial suppliers, even special-purpose real time operating systems. Because of the wide-spread customer use, supplier skill, and development/support experience, the reliability of such products is typically better than could be attained by developing a custom operating system.

OTS software or any reused software should be analyzed to consider the above concerns prior to use. The following general guidance is derived from the references [NUREG6421], [NATO96], [NATO97], [JA1005], and [DEFS0055].

6.3.1 SUPPORT TOOLS

Guide 1: All tools used in the software development and/or support should have sufficient reliability assurance to ensure that they do not jeopardize the reliability of the application system.

Guide 2: The reliability requirements for the tools should be deduced from the reliability analysis of the software development process. This analysis should define the required reliability of each tool with respect to its use in the project. Such tools do not necessarily need assurance if they are used only to highlight issues for further investigation. If they are used to confirm behavior, then they should be appropriately reliable.

Guide 3: Process hazards should be identified where the level of safety assurance required is not met by the process and tools and where the limitations of particular tools introduce a specific technical flaw into the process. Appropriate safeguards should be put into place as defense against the identified process hazards such that the complete development process achieves the required safety assurance.

Guide 4: Each tool should be evaluated to determine conformance with the required reliability requirements for the tool's proposed use in the development of the system/software application.

Guide 5: In addition to conformance to reliability requirements, tool selection criteria should include other quality criteria such as usability, interoperability, stability, commercial availability, vendor maintenance support and familiarity to the Design Team.

6.3.2 USED IN NEW OR MODIFIED SYSTEMS

Guide 1: Reliability analysis of the system/software application should be conducted to demonstrate that there are no adverse reliability implications resulting from the use of the previously developed software in the new system.

Guide 2: Previously developed software should conform to the reliability requirements and general software engineering practices required for the system/software application; or, an appropriate mitigation strategy should be demonstrated to have been implemented.

Guide 3: Reverse engineering and/or V&V activities should be carried out on any previously developed software to be used in the new system if evidence does not exist that it has been produced to the requirements of the general software engineering practices required for the system/software application.

Guide 4: The extent of the reverse engineering and V&V activities should take into account the rigor of the previously developed software's development process, the extent of application functionality provided by the previously developed software, and the previously developed software's in-service history.

6.3.3 DURING IN-SERVICE

Guide 1: In-service history should only be taken into account as evidence of the reliability of the previously developed software where reliable data exists relating to in-service usage and failure rates.

Guide 2: All changes to previously developed software made as part of its incorporation in the system/software application should be to the requirements of the general software engineering practices required for the system/software application.

Guide 3: All previously developed software incorporated in the system/software application should be identified in the Software Reliability Case.

Guide 4: Justification that the previously developed software will not adversely affect the reliability of the final system/software application should be provided in the Software Reliability Case.

Guide 5: All previously developed software incorporated in the final application should be provided with documentation equivalent to the general software engineering practices required for the system/software application.

NOTE—When changes are made to the previously developed software, there may be conflicts between the practices of the previously developed software and the general software engineering practices of the required application. In such cases, careful consideration should be made as to the benefits of deviation from versus consistency with the practices of the previously developed software.

6.3.4 DOCUMENTATION

The OTS software documentation appropriate to support evidence of its general software engineering practices and level of reliability is dependent upon how the OTS software is used, criticality of the system/software application, and expected responsibilities for support of the OTS software by the supplier.

Although the OTS software guidance in both [NUREG6421] and [DEFS0055] is targeted to software associated with safety systems, the guidance is really applicable to any OTS software. This guidance targets good software engineering practices augmented by safety-specific practices appropriate to the required integrity level. By including reliability-specific requirements and practices, the guidance would be just as appropriate for addressing OTS software reliability concerns as well as safety concerns. Adding security-specific practices to the mix would allow for a complete surety approach to OTS software.

As a minimum, the documentation should be artifacts derived from the activities such as suggested in the guidance of 6.3.1, 6.3.2, and 6.3.3. Most of this documentation is supplementary to the software reliability plan and software reliability case information and should take a similar format for consistency. Additional information specific to OTS software might include a supplier survey, escrow and maintenance contracts, training materials, and user guides.

6.4 Data Collection and Repositories

While there are numerous approaches to data collection, there exists an inherent set of data and metrics that an organization should collect to improve the quality of their products as well as the processes used to create them. Generally speaking, there are three groups of metrics to be monitored for trends: product metrics, process metrics, and resource metrics.

6.4.1 Organizational Responsibilities

A product development team should be established to assess the validity of the data collected, track fault resolution, provide statistical analyses, and assure that the necessary corrective action is implemented whenever necessary. The members of this team should consist of representatives from the appropriate Design teams, a Quality Assurance representative, a Reliability engineer, a program planning and scheduling person, and a finance representative. The group should appoint a leader to schedule and conduct team meetings, provide the team/management interface, assign necessary action items, publish team decisions, and record meeting minutes. An appropriate reliability data collection capability should be based on the customer's needs initially identified during the "Determine Customer Requirements" task activity and evolved as necessary as tasks are conducted to ensure reliability requirements are met and demonstrate to the customer that the requirements have been met.

6.4.2 Data Collection

Certification has become increasingly important in the contractor role and in turn has lead to increased emphasis on data collection processes. Collecting the appropriate types of data allows the contractor to analytically derive the metrics necessary to evaluate the product, such as test stop time, release quality as well as the effectiveness of the quality of properly collected data. For in-service data, it will be necessary to distinguish failures caused by software, hardware, a combination of hardware and software, software data and/or instructions, and other potential sources such as operator error. The data collection effort should include training the individuals responsible for the collection and entry of data into the database as well as obtaining management buy-in. Once an effective data collection system becomes established, it will be a necessary tool in the product evaluation process. It will also be necessary to plan for and manage the sustainment of this data collection system.

6.4.3 MEASUREMENT AND METRICS

A complementary set of software reliability measurements and metrics should be collected, integrated, and analyzed throughout the software life cycle in order to provide a comprehensive and ongoing assessment of software reliability. These measurements and metrics should be defined during the pre-development phase. Goal and acceptance criteria for actual results should be established for each measurement and metric using techniques such as Goal, Question, Metric (C.1.6). Metrics should be selected that will: (1) measure different aspects of the product and the process and resources used to develop it; and (2) be used during each of the life cycle phases. The correlation between actual results and stated goals and/or regulatory and contractual requirements should be monitored during each life cycle phase and for each category and subcategory of metrics. This will permit corrective action to be taken in a timely and cost effective manner. Metrics should be chosen based on the size, scope, duration, complexity, and risk of a project, as appropriate for the stated reliability goal. The metrics used will vary depending on the needs of the project.

6.4.3.1 Product Metrics

Product metrics provide a quantitative assessment of whether or not a project is on target for meeting stated software reliability goals. As defined in [IEEE982-1], there are four categories of product metrics: completeness and consistency; complexity; error, fault and failure; and reliability growth and prediction. Completeness and consistency metrics indicate whether or not customer requirements are being met. Complexity metrics indicate if a software implementation is overly complex, which will complicate validation and verification activities, as well as future maintainability. Error, fault, and failure metrics record the number, type, and severity of defects found during the different life cycle phases. Reliability growth and prediction metrics are used to predict future behavior once a product is fielded. An appropriate set of product metrics should be selected from these four categories, based on the customer's needs. The results of collecting, analyzing, and interpreting product metrics should be recorded in the Software Reliability Case.

6.4.3.2 Process Metrics

Process metrics provide both a quantitative and a qualitative assessment of the integrity of the process(es) used to develop a software product. As explained in Part 3 of [IEC61508], the degree of rigor applied to a development process increases in proportion to the level of software reliability required; i.e., the higher the software reliability goal, the more stringent the process controls. As defined in [IEEE982-1], there are two categories of process metrics: sub/process effectiveness and management control. Sub/process effectiveness metrics analyze the results of reviews, audits, and inspections, and the use of static and dynamic analysis techniques. Management control metrics measure the intervals between the time when an error was introduced, detected, and removed. An appropriate set of process metrics should be selected from these two categories, based on the customer's needs. The results of collecting, analyzing, and interpreting process metrics should be recorded in the Software Reliability Case.

6.4.3.3 People/Resource Metrics

People/resource metrics provide a qualitative assessment of the appropriateness of the people and resources used to develop a product and enforce a process. The four categories of people/resource metrics are competency, schedule reality, development environment, and human computer interface (HCI) issues. Competency metrics evaluate whether or not the project staff have the appropriate education, experience, and certification to develop a software system with the stated reliability goal. Schedule reality metrics assess whether an appropriate mix of people and resources are being applied at the right time during the project. Development environment metrics evaluate whether appropriate hardware and software platforms and automated tools are being used on the project. HCI metrics evaluate whether issues related to domain knowledge, operational profiles, and so forth have been adequately addressed. An appropriate set of people/resource metrics should be selected from these four categories, based on the customer's needs. The results of collecting, analyzing, and interpreting people/resource metrics should be recorded in the Software Reliability Case.

6.4.4 FAILURE INCIDENT REPORT FORM

The example database form, shown in Figure 9 illustrates a failure incident report that would be part of a FRACAS (C.4.2) database application tool. This form includes problem description, failure identification, failure corrective action information and configuration management change board decisions. In addition, logistics information is included to uniquely identify the failed computational component and the associated software product and module fault information. Defining the associated queries, data sorts, and reports would be based on the functional requirements of the software, i.e., safety critical, flight critical, security of data, and so forth.

6.4.5 DATA REPOSITORIES

There are some limited existing reliability data base repositories that provide an indication of the type of reliability data that might be important to collect in order to support reliability analysis. An example of such data from the AIAA repository is defined in reference [AIAAR013] and summarized in Table 10. The Data and Analysis Center for Software (DACS) and Reliability Analysis Center (RAC) also provide some support for reliability data repositories as well as other software engineering and reliability information resources.

RAC is a source for software reliability information and references. It maintains extensive links to external reliability resources as well as provides its own sources of reliability reports and tools. DACS provides a software reliability source book [DACS02] that includes: an overview of software reliability, definitions, basic statistical concepts & methods, selection & use of reliability/quality metrics, software design approaches, reliability allocation, prediction and estimation, analytical reliability techniques, and reliability test techniques/processes.

The reference [BASILI02] identifies a Center for Empirically-Based Software Engineering (CeBASE, http://www.CeBASE.org). CeBASE accumulates empirical models in order to provide validated guidelines for selecting techniques and models, recommend areas for research, and support software engineering education. One CeBASE objective is to validate empirical software engineering assumptions. Software metric/reliability data will support that objective. Guidelines for specific data to collect may be an important future result from this effort.

TABLE 10—EXAMPLE RELIABILITY DATA FOR THE AIAA REPOSITORY

Data Category	Data Item
Project Data	Information to identify and characterize each system and effort that generates data stored in the database.
	-life cycle activities and schedule information for each activity -Development environment characteristics
Component Data	Information for each system component, e.g., module
	-size in source lines of code, function points -source language -complexity measure
Dynamic Failure Data	Information for each failure recorded
	-activity being performed (test, operation, maintenance) -date and time of failure -severity of the failure (critical, major, minor) -type of failure (interface, logic, data, user input/output) -failure detection method (inspection, test) -component(s) where root cause fault(s) found - CPU hours since last failure; number of test cases since last failure; wall clock hours since last failure; test hours per test interval and number of failure detected in each interval; test labor hours since last failure
Fault Correction Data	Information for each failure corrected with a software change.
	-date and time the software change/fix was available -source of fix (requirements, design, code) -type of change (correction, enhancement, adaptation, no change) -labor hours required for correction (analysis, change, test) -CPU hours required for the change/fix; number of runs required to make the fix; wall clock hours used to make the correction
Lessons Learned Data	Information about the project (development/support) and contacts
	-corporate knowledge of the software development, testing, maintenance support -lessons learned and contact person with knowledge about the project

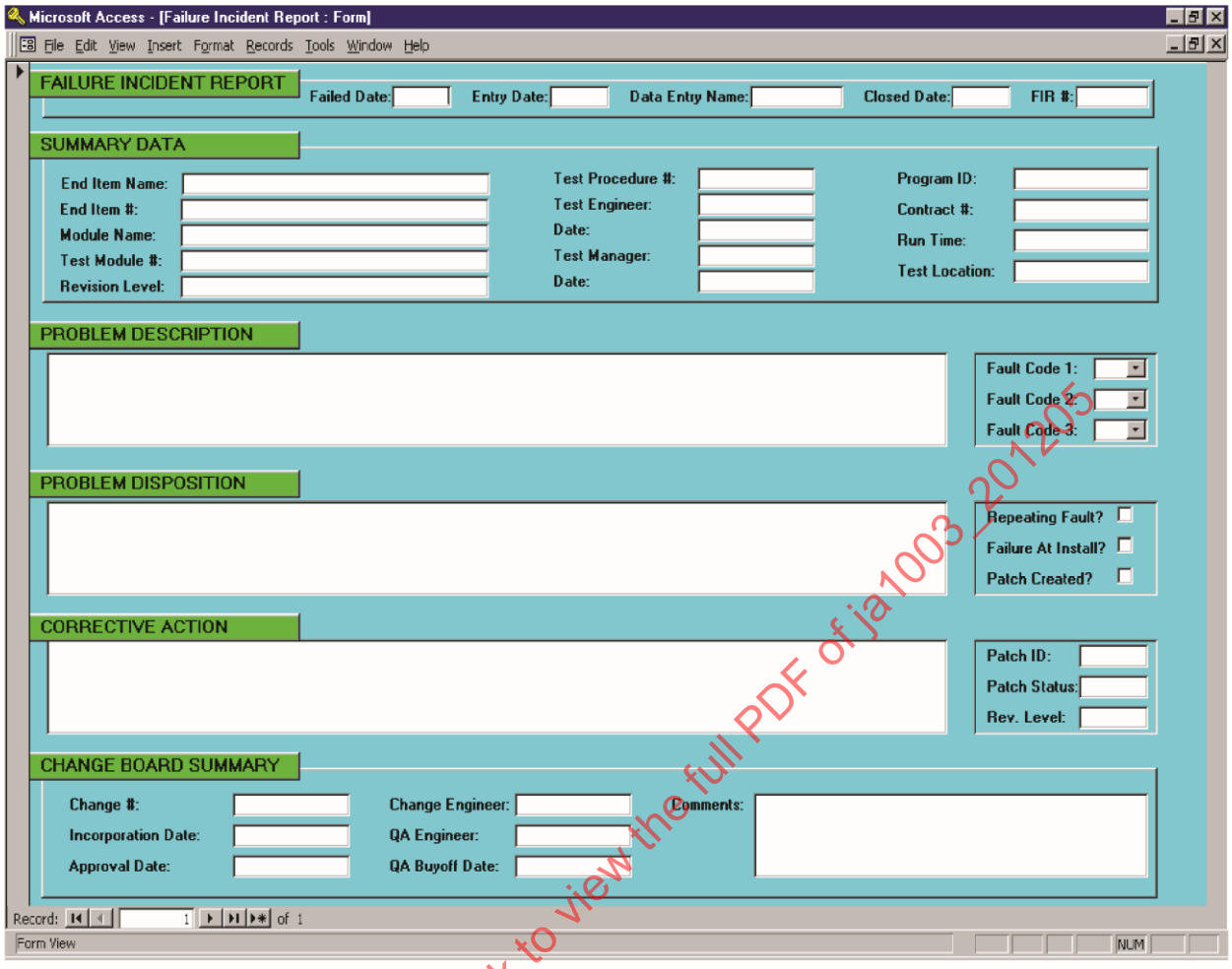

FIGURE 9—EXAMPLE FAILURE INCIDENT REPORT FORM

PREPARED BY THE SAE G-11 RELIABILITY, MAINTAINABILITY, SUPPORTABILITY AND LOGISTICS (RMSL) SOFTWARE COMMITTEE

REAFFIRMED BY THE SAE G-11 RMSL SOFTWARE COMMITTEE

APPENDIX A
RELATIONSHIP TO EXISTING STANDARDS AND GUIDELINES

There are many standards and guidelines from military and industry sources for which this guideline should be an important resource. This guideline, JA1003, may provide directly aligned guidelines such as for [JA1002]. JA1003 may provide supplementary or complementary guidelines, such as for [DO178B] or [DEFS0042-2]. JA1003 may integrate well with a software engineering standard such as [ISO12207] or [IEEE12207]. JA1003 may integrate with a system engineering process standard such as [CMMI] or implement software-specific reliability concerns within a system standard, such as [JA1000] or [ARMP-1]. In some cases, the referenced guidelines such as in [AIAAR013] and [BSI5760-P8] provide more detailed information that would support implementing methods/techniques summarized in JA1003. In addition, there is parallel guidance that exists within many standards and guidelines in the strongly related areas of safety, security, maintainability, supportability, and dependability such as [DEFS0055], [JA1010], and [JA1005]. A selected set of standards and guidelines is identified in Table A1 along with a variety of characteristics that illustrate some of these relationships to this JA1003 guide.

Many of the standards and guidelines referenced in this document are discussed in some detail in reference [HERRM99]. The publication [PRIME97] is intended to provide capsule summaries of the most pertinent US and international commercial and government specifications, standards and handbooks dealing with reliability, maintainability, availability and dependability. It is intended for program managers and other individuals who need a concise overview of the most important applicable documents available in the field. The publication enables readers to determine the applicability of the documents without having to obtain them first. In addition, Internet sites exist (see 2.2) that have a world-wide search capability for reliability standards and standards organizations.

TABLE A1—CHARACTERIZATION OF STANDARDS AND RELATIONSHIP TO SOFTWARE RELIABILITY

Organization	Standard/Guideline	JA1003 Relationship	Primary Application Area	Relationship of JA1003 to Standard/Guideline
SAE	JA1000	System Reliability Standard	Aerospace and Ground Vehicles	Directly aligned
	JA1000-1	System Reliability Guide	Aerospace and Ground Vehicles	Directly aligned
	JA1010	System Maintainability Standard	Aerospace and Ground Vehicles	Parallel
	JA1010-1	System Maintainability Guide	Aerospace and Ground Vehicles	Parallel
	JA1002	Software Reliability Standard	Aerospace and Ground Vehicles	Directly aligned
	JA1003	Software Reliability Guide	Aerospace and Ground Vehicles	Same
	JA1004	Software Supportability Standard	Aerospace and Ground Vehicles	Parallel
	JA1005	Software Supportability Guide	Aerospace and Ground Vehicles	Parallel
	JA1006	Software Support Concept Guide	Aerospace and Ground Vehicles	Parallel
ISO	ISO 12207	Software Engineering Standard	Universal	Supplementary support
ISO/IEC	ISO 15288	System Engineering Process Guide	Information Technology	Supplementary support
IEC	IEC 61713(Draft)	Software Dependability Guide	Universal	Parallel
	IEC 61719(Draft)	Software Dependability Measurement Guide	Universal	Parallel
	IEC 61508	System Electronics Safety Standard	Electronic systems	Supplementary support
	IEC 61511-1	System Electronics Safety Standard	Process Industry Sector	Supplementary support
NATO	ARMP-1,4,6,7	System R&M Standard and Guidance	NATO Collaborative Procurement	Directly aligned
IEEE	IEEE 12207	System Software Engineering Standard	Universal	Supplementary support
	IEEE 982.1/2	Software Measurement Standard	Universal	Framework for detailed methods
	IEEE Std. 1228	Software Safety Standard	Universal	Parallel
	IEEE Std. 1413	Reliability Prediction & Assessment Guide	Electronic Systems and Equipment	Framework for method assessment
UK MOD	DefStan 00-42	System/Software R&M Standard	UK MOD Procurement	Directly aligned
	DefStan 00-55	System/Software Safety Standard	UK MOD Procurement	Parallel
	DefStan 00-60	System Logistics Standard	UK MOD Procurement	Parallel
UK BSI	BS5760	System/Software Reliability Process & Product Assessment Guide	Universal	Framework for detailed methods
AIAA	AIAA/R-013	Software Reliability Recommended Methodology and Practices Guide	Universal	Framework for detailed methods
FAA	DO178B	Software Engineering/Safety Standard	Airborne Systems & Equipment Certification	Supplementary support
SEI	CMMI	System/Software Engineering Process Framework	Universal	Supplementary support
RAC	[ROME97]	System/Software Reliability Assurance Guide	Universal	Framework for detailed methods
MISRA	MISRA-VBS	Software Engineering Guidelines	Motor Vehicles	Supplementary support

APPENDIX B
EXAMPLE PLAN AND CASE OUTLINES

B.1 Software Reliability Plan Thematic Outline

1. MANAGING THE SOFTWARE RELIABILITY PROGRAM ACTIVITIES

    1.1 Define purpose, scope of plan and program, reliability goals and objectives

    1.2 Nomenclature and project references

    1.3 Program management functions: responsibility, authority, interaction between system and software reliability programs; customer interaction/involvement; risk management

    1.4 Resources needed, including personnel and equipment

    1.5 Schedule

    1.6 Training

    1.7 Subcontract Management

    1.8 Plan approval and maintenance

2. PERFORMING SOFTWARE RELIABILITY PROGRAM ACTIVITIES

    2.1 Determine Customer Requirements

        2.1.1 Establish supplier-customer dialogue

        2.1.2 Identify operational conditions of use

        2.1.3 Define in-service conditions of support

        2.1.4 Establish metrics: goals, assumptions and claims, and expected evidence

        2.1.5 Develop plan

        2.1.6 Document pre-development case evidence

    2.2 Meet Customer Requirements

        2.2.1 Define life cycle model and interaction with system engineering

        2.2.2 Design reliable solution

        2.2.3 Implement reliable solution

        2.2.4 Verify and validate solution is reliable

        2.2.5 Assess and manage reliability risk

2.2.6 Document development case evidence

2.3 Demonstrate Customer Requirements

2.3.1 Qualify/certify the product and process

2.3.2 Establish process controls

2.2.3 Transition to operational environment

2.2.4 Train end-users, operations and support staff

2.3.5 Pursue continuous improvement

2.3.6 Establish data collection and reporting

2.3.7 Document post-development case evidence for in-service use

3. DOCUMENTING SOFTWARE RELIABILITY PROGRAM ACTIVITIES

3.1 Lifecycle practices

3.2 Software reliability case file of evidence

B.2 Software Reliability Case Thematic Outline:

1. SYSTEM CONTEXT DESCRIPTION

1.1 Equipment and system boundary

1.2 Usage and operating environment

1.3 Interfaces, build standard, and configuration

1.4 Operating and maintenance personnel qualification

1.5 Support/maintenance policy as it applies to sustained reliability

2. SOFTWARE RELIABILITY GOALS, OBJECTIVES, REQUIREMENTS

2.1 What they are, overall and for partitions

2.2 How were they derived, apportioned to software and partitions

2.3 Relation to system reliability goals

2.4 Regulatory and/or contractual requirements

2.5 Areas of potential risk

2.6 Summary of reliability strategy

3. ASSUMPTIONS AND CLAIMS

   3.1 Assumptions: agreed upon constraints and basis for claims

   3.2 Claims: agreed upon validation and certification criteria; traced to requirements

4. EVIDENCE

   For each of the phases Pre-development/Development/Post-development/In-Service provide evidence related to:

   4.1 Process activities and evaluations that demonstrate achievement of software reliability claims

   4.2 Product activities, characteristics, and analyses that demonstrate achievement of software reliability claims

   4.3 Qualifications of people and resources conducting the activities for which evidence is being provided

5. CONCLUSION/RECOMMENDATION

   5.1 Summary of reliability requirements, claims, and actual evidence provided

   5.2 Limitations based on system use boundaries, environment and support constraints

   5.3 Recommendations related to certification, qualification, warranty

6. CERTIFICATION RECORDS

   6.1 Reference record of all acceptance warranty, certification, qualification activities and results

APPENDIX C
TASK ACTIVITIES, METHODS, AND TECHNIQUES

Table C1 lists current techniques which can be used to achieve and assess different aspects of software reliability throughout the life cycle phases defined in reference [JA 1002]: pre-development (concept & feasibility), development (design &production), and in-service. As indicated in Table C1, many of the techniques can be used in multiple life cycle phases to implement software engineering/reliability task activities such as described in Section 5. In particular, it is important to note that many techniques used during development can also be used during the in-service phase to investigate an accident/incident and/or determine why a system did not achieve its stated reliability goals. There are other existing techniques and new ones being developed that may be more suitable for a specific application. The basic guidelines presented in this document provide a valid framework for selecting the most appropriate techniques depending on the application and customer requirements.

Following Table C1, a one-page (or more) description of each technique is provided. The intent is for each project team to choose a complementary set of analysis, design and verification techniques that are appropriate for their specific project. These choices should be listed and explained in the Software Reliability Plan. The results obtained from using these techniques should be included as part of the evidence in the Software Reliability Case.

The descriptions of the software reliability techniques listed in Table C1 follow a similar format to that used in reference [JA1000-1]. It is not possible to provide all information related to each technique nor all pertinent references, but these summary pages should be helpful in getting an initial view of the technique and locating more details about the technique. The information describing each technique is organized as follows:

a. Purpose: What is achieved by using the technique;

b. Description: Main features of the technique;

c. Benefits: Value the method adds to the product development and assessment process;

d. Case Evidence: Results of applying the technique that might be included in the software reliability case;

e. Limitations: Factors that may limit the use of the technique and/or effect the interpretation of the results obtained; and

f. References: Sources for more information about the technique.

TABLE C1—TECHNIQUES TO ACHIEVE AND ASSESS SOFTWARE RELIABILITY BY LIFE CYCLE PHASE

TECHNIQUES	Section	LIFE CYCLE PHASES		
		Concept & Feasibility	Design & Production	In-Service
C.1 Analysis Techniques				
change impact analysis	C.1.1		x	x
common cause failure analysis	C.1.2	x	x	x
defect removal efficiency	C.1.3	x	x	x
design of experiment	C.1.4	x	x	x
formal scenario analysis	C.1.5	x	x	x
goal-question-metric (GQM)	C.1.6	x		x
hazard analysis	C.1.7	x		x
pareto analysis (execution time, failure sources)	C.1.8		x	x
probabilistic methods	C.1.9		x	x
Pugh selection	C.1.10	x	x	x
quality function deployment (QFD)	C.1.11	x	x	x
reliability allocation	C.1.12	x	x	
reliability block diagrams	C.1.13		x	x
reliability prediction modeling	C.1.14	x	x	
response time, memory, constraint analysis	C.1.15		x	x
six sigma	C.1.16	x	x	x
sneak circuit analysis	C.1.17		x	x
software failure modes, effects, criticality analysis (SFMECA)	C.1.18	x	x	x
software fault tree analysis (SFTA)	C.1.19	x	x	x
software reliability engineering (SRE-Musa)	C.1.20	x	x	x
statistical analysis	C.1.21		x	x

TECHNIQUES	Section	LIFE CYCLE PHASES		
		Concept & Feasibility	Design & Production	In-Service
C.2 Design Techniques				
design by contract	C.2.1	x	x	x
fault tolerant design	C.2.2		x	x
formal methods/languages	C.2.3	x	x	
independence, isolation, inoperability, incompatibility (I4)	C.2.4	x	x	x
mistake/error proofing	C.2.5	x	x	x
Petri nets	C.2.6		x	
software integrity checks	C.2.7		x	x
C.3 Verification Techniques				
boundary value analysis	C.3.1		x	x
cleanroom	C.3.2		x	
coverage analysis	C.3.3		x	x
dynamic test methods	C.3.4		x	x
formal in-process reviews (Fagan software inspections)	C.3.5		x	x
operational profile	C.3.6		x	x
peer reviews	C.3.7		x	x
reliability bench marking	C.3.8		x	
reliability estimation modeling	C.3.9		x	x
root cause analysis	C.3.10		x	x
testability analysis, fault injection, failure assertion	C.3.11		x	x
C.4 Management Techniques				
configuration management	C.4.1		x	x
failure reporting and corrective action system (FRACAS)	C.4.2		x	x
life cycle process standard	C.4.3	x	x	x
process assessment	C.4.4	x	x	x
requirements management	C.4.5	x	x	x
risk management	C.4.6	x	x	x

C.1 Analysis Techniques

C.1.1 Change Impact Analysis

a. Purpose: Analyze apriori the potential local and global effects of changing requirements, design, data structures, and/or interfaces on system performance and reliability.

b. Description: Changing or introducing new requirements or design features may have a ripple effect on the current system. A change or fix may be applied to one part of a system with detrimental or unforeseen consequences on another part Change impact analysis evaluates the extent and impact of proposed changes by examining which requirements and design components are interdependent. Change impact analysis can also be used to support analysis of alternatives, by highlighting which alternative can be implemented most efficiently, and to identify the extent of reverification and revalidation needed (regression testing, C.3.4). Change impact analysis can be used to determine how measures such as interdependency, coupling, cohesiveness, and inheritance can be improved to meet goals (14, C.2.4).

c. Benefits: The potential for uncovering or introducing new errors when implementing changes is minimized.

d. Case Evidence: Design architecture analysis for interdependency, coupling, cohesiveness, and inheritance areas of improvement to meet measurement goals; proactive evidence of defect prevention during development and support by reducing required reverification and revalidation efforts when changes are made; evidence of improvement in reliability effectiveness.

e. Limitations: The scope of the analysis determines its effectiveness.

f. References:

1. Arnold, R. and Bohner, S., "Software Change Impact Analysis," IEEE Computer Society Press, 1996.

2. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

3. Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

4. System Safety Society, System Safety Analysis Handbook, July 1993.

C.1.2 Common Cause Failure Analysis

a. Purpose: Identify failure scenarios in which two or more events could occur as the result of a common design defect.

b. Description: Common cause failure analysis seeks to identify intermediate and root causes of potential failure modes. The results of common cause failure analysis is often documented graphically by event trees. This information is analyzed to determine failures that could result from common design defects, hardware failures or anomalies and to propose the requisite mitigating actions, such as the need for I4 implementations (C.2.4), or other such strategies. Common cause failure analysis should include both hardware and software components. See also root cause analysis (C.3.10), software FMECA (C.1.18), and software FTA (C.1.19).

c. Benefits: Common cause failure analysis results in a more robust design architecture.

d. Case Evidence: Graphical event trees or other similar evidence that specific failure modes have been identified and resolved with mitigating actions; proactive evidence that design architecture is reliable; supports safety and security system/software FMECA and system/software FTA.

e. Limitations: The extent to which the analysis is carried out, i.e., how far back intermediate and root causes are identified, determines its effectiveness.

f. References:

1. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

2. Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

3. System Safety Society, System Safety Analysis Handbook, July 1993.

C.1.3 Defect Removal Efficiency

a. Purpose: Measures the percentage of software defects removed prior to delivery of the software to clients.

b. Description: Defects can be inserted into software artifacts during any life cycle activity, including the support phase. It is well-documented that it is more cost-efficient to remove defects earlier in the product cycle than later. The following table illustrates the concept of measuring when defects are introduced, when defects are removed, and the "efficiency" with which defects are removed at any given stage. The "efficiency" is the percentage of total defects removed at any stage compared with the total defects that are documented to exist at that stage. Defects are counted through operational use and support activities.

TABLE C2—DEFECT ORIGIN AND REMOVAL EFFICIENCY METRICS

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Document	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
Total	5.00	85%	0.75

Formal In-Process Reviews (C.3.5) is one way to determine information needed to compute the defect removal efficiency for requirements, design, implementation, and test planning reviews. Unit, integration, and system testing provide additional defect removal information prior to customer release. Customer operational defects during a specified version release provide the basis for the overall defect removal efficiency. The defect removal efficiency may also provide a quality level measure in terms of a defect density measure.

TABLE C3—DEFECT POTENTIAL AND REMOVAL EFFICIENCY BY QUALITY LEVEL

Quality Level	Defect Potential (min, max, avg)	Removal Efficiency (min, max, avg)	Delivered defects per function point
1	(~3.0, >15, 5.0)	(<70%, >95%, 85%)	~0.75
2	(~3.0, >12, 4.8)	(<70%, >96%, 87%)	~0.60
3	(~2.5, > 9, 4.3)	(<75%, >97%, 89%)	~0.47
4	(~2.3, > 6, 3.8)	(<80%, >99%, 94%)	~0.20
5	(~2.0, > 5, 3.5)	(<90%, >99%, 97%)	~0.10

The software capability maturity as measured by the Capability Maturity Model (C.4.4) has been related to defect density measures as illustrated in the table below (delivered defects per function point):

TABLE C4—DEFECT DENSITY RANGES BY SEI CMM LEVEL

SEI Level	Minimum	Average	Maximum
1	0.150	0.750	4.500
2	0.120	0.624	3.600
3	0.075	0.473	2.250
4	0.023	0.228	1.200
5	0.002	0.105	0.500

c. Benefits: Defect removal efficiency provides an early focus on defect prevention and removal, a measure of design reliability, and a measure of engineering processes effectiveness. Integrated with Root Cause Analysis (C.3.10) and process improvement assessments (C.4.4), software reliability can be improved.

d. Case Evidence: Defect removal efficiency measure for requirements, design, implementation, test, support; defect density measures; estimate of SEI Level

e. Limitations: The categorization of "defects" in terms of severity level, type, and source may be difficult. Customer classification and reporting of defects may not be possible for some software products.

f. References:

1. Jones, Capers, Assessment and Control of Software Risks, Prentice Hall, 1994.

2. Jones, Capers, Applied Software Measurement: Assuring Productivity and Quality, McGraw-Hill, 1996.

3. Jones, Capers, Software Assessments, Benchmarks, and Best Practices, Addison Wesley, 2000.

4. Kan, S.H., Metrics and Models in Software Quality Engineering, Addison-Wesley Publishing Co., 1995.

C.1.4 Design of Experiment (DOE)

a. Purpose: Process of planning an experiment so that appropriate data will be collected and then analyzed by statistical methods resulting in valid and objective conclusions.

b. Description: Classical Design Of Experiment (DOE) for software is based on establishing a testing strategy to optimize the test coverage effectiveness. "Tests" may be of the software product or may be external experiment tests used to validate a software product. For example, given certain boundary conditions, one might select test parameter values inside, on, and outside the boundary. With many parameters and multiple input possibilities, it is cost effective to conduct sensitivity, convergence, and other tests/experiments to statistically determine the software's range of validity. Deciding which experiments to select is a classic DOE problem. One DOE application of particular importance is for software that implements a physics thermal/mechanical/electrical model or some coupled model combination. The DOE establishes software verification testing and experimental validation strategies. Analytic, semi-analytic, and manufactured solutions will drive the verification testing. Physics experiments will drive the validation testing to determine the capability of the software (implemented model, algorithm, numerical methods) to accurately represent application reality. A thorough analysis of the requirements and understanding of the usage of the software (e.g., see C.1.5 and C.3.6) is needed to prepare the parameter-level test/experiment tables and test plans. The standard DOE process, with some interpretation for software application, includes the following steps:

1. Recognition of and statement of the problem: relate to the real-world application being modeled by the software, the parameters/physics/algorithms used in the model, and the software implementation;

2. Choice of factors and levels: relate the factors to boundary conditions, input parameters, model convergence strategies, known analytic solutions, and application uncertainties;

3. Selection of a response variable: relate the response variable to desired model results;

4. Choice of experimental design: array type; replication, randomization, and blocking; sample size

5. Performance of the experiment: conduct of software testing, physics experiments, and so forth;

6. Data analysis: statistical determination whether the tests/experiments are adequate for decisions; and

7. Conclusions and recommendations: decisions based on software verification and/or validation results

Automotive, telecommunication and defense industries report big productivity improvements to traditional testing methods due to two factors: (1) focus on the software usage, and (2) use of a mathematically sound way to span the entire operating domain with a minimum set of test cases.

c. Benefits: Large savings in testing time and cost. For modeling and simulation software, the quantitative measures of the software predictive accuracy, uncertainty of model parameters, and software implementation error can be estimated. These measures provide evidence of "reliable" results for specific applications.

d. Case Evidence: Test coverage measures; test cost-effectiveness measures; verification/validation evidence that software meets its requirements; predictive accuracy of software and uncertainty measure of results.

e. Limitations: Training (method and domain of knowledge) is required to develop and analyze an experimental design; selection of proper experimental factors is critical to successful experimentation, and may take significant sensitivity analysis to determine which factors are the most significant.

f. References:

1. Box, G.E.P., Hunter, W.G., and Hunter, J.S., Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building, John Wiley & Sons, 1978.

2. Phadke, M.S., Quality Engineering Using Robust Design, Prentice Hall, November 1989.

3. Phadke, M.S., "Planning Efficient Software Tests," CrossTalk Journal of Defense Software Engineering, October 1997, pp 11-15.

C.1.5 Formal Scenario Analysis

a. Purpose: Define a quantitative characterization of how the software will be used from which test cases can be developed.

b. Description: Formal scenario analysis develops a scenario-based test model from the analysis of operational scenarios, user-views and events. Scenarios are defined as an ordered sequence of events which accomplishes a functional requirement specified by an end-user. User views are defined as a set of system conditions specific to a class of users. Events are defined as particular stimuli that change a system state and/or trigger another event. Scenarios are recorded in a formalized tree notation, use case, or a structure similar to that used for finite state machines. Formal scenario analysis has similar objective as operational profiles (C.3.6), although the term "operational profile" has a more specific interpretation in its use as part of the Software Reliability Engineering (C.1.20) approach. The results of this technique support other techniques such as Design of Experiment (C.1.4) and various dynamic test methods (C.3.4) such as regression testing, stress testing, and usability testing. In addition, this technique can be used to develop support scenarios that might uncover potential failures such as in software installation/loading or initialization that would substantially affect the reliable operation of the software.

c. Benefits: Formal scenario analysis is useful for identifying deadlock, nondeterministic conditions, incorrect sequences, incorrect initial and terminating states, and errors caused by an incomplete understanding of the domain knowledge.

d. Case Evidence: Supports test coverage analysis measures; analytical evidence of customer requirements to test mapping.

e. Limitations: Formal scenario analysis is somewhat labor intensive because customers, developers, and end-users are involved.

f. References:

1. Hsia, Pei, "Testing the Therac-25: A Formal Scenario Approach, Safety and Reliability for Medical Device Software," Health Industries Manufacturers Association (HIMA) Report No. 95-8, 1995, tab 6.

2. JA1005, SAE Surface Vehicle/Aerospace (JA) Standard 1005, “Software Supportability Program Implementation Guidelines,” Society of Automotive Engineers, 2001.

3. Musa, John D. “Operational Profiles in Software Reliability Engineering,” IEEE Software, March 1993, pages 14-32.

4. Pant, H., Franklin, P. and Everett, W., "A Structured Approach to Improving Software-Reliability Using Operational Profiles," Proceedings of the 1994 Annual Reliability and Maintainability Symposium, Anaheim P. 142-146.

5. Weidenhaupt, Klaus, et al, “Scenarios in System Development: Current Practice,” IEEE Software, March/April 1998, pp 34-45.

C.1.6 Goal, Question, Metric (GQM)

a. Purpose: Goal-Question-Metric (GQM) Paradigm is a means of measuring various software attributes.

b. Description: The GQM paradigm first step is to define measurement goals tailored to the specific needs of an organization. Goals are refined in an operational way into a set of quantifiable questions. Questions imply a specific set of metrics and associated data for collection. This paradigm has been used successfully in several organizations. Typical measurement areas include:

1. Delivered defects and delivered defects per size measure (e.g., function points, source lines of code);
2. Adherence to schedule (e.g., actual vs estimated);
3. Estimation accuracy (e.g., variance with confidence limits);
4. Number of customer problems (e.g., open, closed, rate of closure, by criticality level)
5. Time that problems remain open (e.g., time from receipt to closure); and
6. Software reliability (e.g., failure rate, defect density, probability failure will not occur).

A template devised by Basili and Rombach to define measurement goals is:

"Analyze <object of study> in order to <purpose> with respect to <focus> from the point of view of <point of view>."

where each of the bracketed terms <...> defines the object, purpose, focus, and viewpoint of the measurement goal.

Typical goals related to software reliability might be:

1. Goal: Decrease delivered software defect density from 0.75 per function point to 0.10 per function point within 2 years.
2. Goal: Improve delivered software failure rate per 1000 execution hours from 0.1 to 0.01 within 2 years.
3. Goal: Meet the required delivered software reliability of 0.90 for the time period of 1000 execution hours.

c. Benefits: The GQM method is simple, can be used to related software measurement to organization goals, costs very little to implement, and has been effectively used by many organizations

d. Case Evidence: Metrics related to defects, field problems, defect density, failure rate, engineering process adherence or improvement, or any other such measures that might relate to a basic case evidence claim would be typical case evidence.

e. Limitations: Goals may not be able to be stated in precise enough terms so as to know when the goal has been met; required negotiation may not be possible among the customer, supplier, and certification authority in order to agree on the goal as well as what questions, metrics and supporting data would adequately support the goal - this is similar to being unable to agree on a case evidence claim or the evidence required to support that claim.

f. References:

1. Basili V. R. and Mendonca M. G., "Validation of an Approach for Improving Existing Measurement Frameworks," 484 IEEE Transactions on Software Engineering, vol. 26, no. 6, pp 484-499, June 2000.

2. Basili, V.R. and Rombach, H.D., "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. 14, no. 6, pp. 758-773, June 1988.

3. Basili, V.R. and Weiss, D.M., "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. 10, no. 6, pp. 728-738, November 1984.

C.1.7 Hazard Analysis

a. Purpose: Identify threats (hazards) that will apply to the design of a specified software based system; identify the most safety-critical areas of the software.

b. Description: Conventional hazard analysis considers faults likely to lead to failures based on physical effects. When hazard analysis is to be applied to software-based systems, the problem is more complicated and perhaps less amenable to “conventional skills”, since it is design faults that must be considered and not physical degradation. The typical objectives for hazard analysis include:

1. identifying critical system modules and program sections, i.e., those with most safety relevance;

2. verifying that software required to handle the failure modes identified by systems/subsystems hazard analysis does so effectively;

3. allowing more rigorous methods and controls to be selected and applied to areas of software which are most critical to the safety of the system;

4. identifying and evaluating safety hazards associated with the software, with the aim of either eliminating them or assisting in the reduction of associated risks;

5. identifying failure modes that can lead to an unsafe state and making recommendation for changes; and

6. determining the sequence of inputs which could lead to the software causing an unsafe state and making recommendations for changes.

Examples of some general threats that affect the design of any software based system include:

1. environmental and operating conditions;
2. logic control (real time executive);
3. system function calls;
4. system resources;
5. timing; and
6. software design notations.

Analysis of these and other threats will assist in identifying critical software modules and unsafe failure modes. This knowledge may then be used to change the design to reduce the risk of a hazard occurring or to direct the more rigorous development and fault tolerant design techniques to the most critical areas of software. Software FMECA (C.1.18) and Software FTA (C.1.19) can be classified as hazard analysis techniques. The Falla reference below is an excellent source for descriptions, research results, and software-specific applications of hazard analysis as well as SFMECA and SFTA.

c. Benefits: Provides a systematic approach to identifying hazards that software may create or enhance; through mitigating actions reduce the risk of the identified hazards through software design changes.

d. Case Evidence: Identification of any safety critical part of the software system and evidence of mitigation strategies implemented to reduce the effects of identified hazards.

e. Limitations: Hazard analysis is most effective when conducted within a system safety program that purposefully integrates the software component; this requires specialized personnel resources, additional project cost and schedule effort, and management commitment.

f. References:

1. Falla, Mike, "Results and Achievements from the DTI/EPSRC R&D Programme in Safety Critical Systems," Edited by Mike Falla, Motor Industry Software Reliability Association, November 1996.
2. ISO/IEC 61511-1, Edition 1.0: "Functional safety of electrical/electronic/programmable electronic safety-related systems," International Electrotechnical Commission, 2003.
3. Leveson, Nancy G., Safeware: System Safety and Computers, Addison Wesley Publishing Company, 1995.

C.1.8 Pareto Analysis

a. Purpose: Provide the engineering community with a method for identifying, categorizing, and representing which subsets of problems occur more frequently within the problem domain.

b. Description: Pareto analysis is one of the basic seven quality control tools: check sheets, Pareto charts, Ishikawa diagrams, flow diagrams, histograms, scatter plots, and control charts. Pareto analysis is based on the Pareto Principle (80% of the problems are due to 20% of the possible causes) identified by Vilfredo Pareto, an Italian economist. Since Pareto analysis determines which subsets of problems occur more frequently, it also provides guidance into the prioritization of problems and the efficient allocation of resources to solve the problems. Typically Pareto analysis applied at the system level might identify "software" as a category in which a high frequency of problems (e.g., defects/faults/failures) occurs. Within the "software" category, the problems may be further categorized by Pareto analysis to indicate what types of software problems occur most frequently (or perhaps have the most severe impact), and perhaps the components of the software system in which the problems are occurring. Prioritization of resources can be applied to resolving the problems in the most effective way. Root cause analysis (C.3.10) investigation might lead to process improvement opportunities.

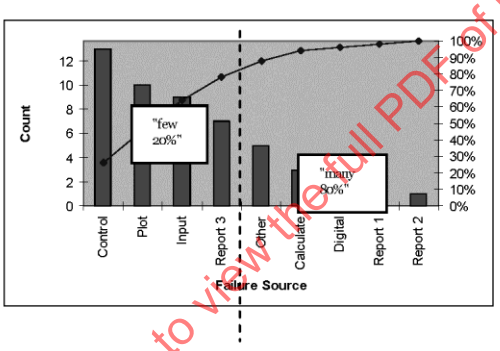

FIGURE C1—EXAMPLE PARETO CHART OF SOFTWARE PROBLEMS

The key elements to effectively apply Pareto analysis techniques includes the following steps:

1. assemble the data to be analyzed and establish a counting rule;
2. add up the total of each item under analysis in accordance with the counting rule;
3. list the items in order of count magnitude, starting with the largest;
4. calculate the count total of all the items, and the percentage that each item represents of the total;
5. construct a bar chart representing categories on one axis and percentage on the other axis in order of percentage from largest to smallest, label the diagram with category titles and percentage marks; and

6. interpret the diagram as to what the categories mean, possible root causes, prioritization, and allocation of resources to resolve the problems represented on the diagram.

c. Benefits: Pareto analysis is simple to use and provides a simple visual communication mechanism between engineers who must implement solutions and management who must decide how to prioritize and allocate limited resources. Tracking results of action by comparing Pareto charts over time is a simple way to show improvement results.

d. Case Evidence: Ordered list of highest frequency events (e.g., modules failing)

e. Limitations: It is possible to draw improper conclusions based on incomplete or inaccurate comparisons of data. One must use common sense to understand what the chart is and is not describing. Certain problem categories may be infrequent, but may have a much larger impact. Further investigation such as root cause analysis may be required to provide more complete information prior to key decisions.

f. References:

1. Juran, J.M, and Godfrey A.B., Juran's Quality Handbook, 5th Edition, McGraw Hill, December 1998.

2. Deming, W.E. and Walton, M., The Deming Management Method, Perigee Publishing Co., January 1988.

C.1.9 Probabilistic Methods

a. Purpose: Provide the ability to predict an outcome accurately.

b. Description: Probabilistic methods are used to model uncertainties, develop probabilistic process models, compute probabilities, estimate risk, identify most likely outcomes, provide sensitivity measures, and identify key drivers. Probabilistic technology works through the combination of predictive models: deterministic process, process variable, and uncertainty. A deterministic process model is a mathematical representation of how an event works. It may be behavioral, process, physics, and/or rule based. A process variable model is a mathematical representation of the statistical behavior of the variables that enter the deterministic process model to predict the outcome of the process, including factors associated with uncertainties inherent in these variables. An uncertainty model is a mathematical representation of the uncertainties that could potentially influence the outcome and are not considered as part of the variable models. The general illustration of how probabilistic methods work is shown below.

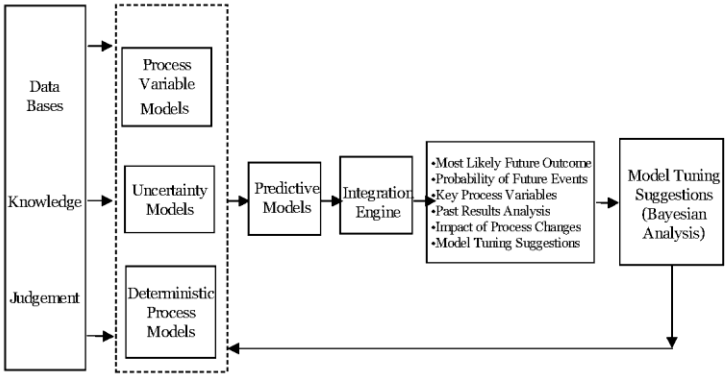

FIGURE C2—EXAMPLE PROBABILISTIC METHODS WORK FLOW

Such probabilistic methods can be used in many ways to support a software reliability program. For example, if the software itself is a large scale implementation of a physics model, then the verification and validation of the software implementation could apply probabilistic methods to determine the predictive validity of the physics model as implemented in software. The software's "predictive accuracy" would reflect its reliability under specific application conditions. Software reliability prediction models (C.1.14) and estimation models (C.3.9) could be analyzed using probabilistic methods and probabilistic testing (C.3.4) to determine the accuracy of those models and applicability for use as a decision tool within a software reliability program.

c. Benefits: Quantify reliability, failure probability, risk, liability, and safety measures; develop optimal strategies for testing, warranties, development/support cycle time; identity and manage uncertainties.

d. Case Evidence: Key variables in reliability models can be identified; reason for reliability prediction; software reliability prediction model accuracy; software reliability estimation model accuracy; evaluation of actual performance against the model

e. Limitations: Probabilistic methods usually require expert users/analysts to interpret the results and a variety of automated tools; existence of necessary parameter data for accuracy analyses may not be available; complexity of approach may make it difficult for non-expert decision makers to accept the probabilistic results.

f. References:

1. Khalessi, M.R., "Probabilistic Technology Description Whitepaper," Prediction Probe, 2002 http://www.predictionprobe.com/

2. AIR 5109, "Applications of Probabilistic Methods," Society of Automotive Engineers, February 2002.

3. AIR 5080, "Integration of Probabilistic Methods into the Design Process," Society of Automotive Engineers, January 1997.

C.1.10 Pugh Selection

a. Purpose: Select the best alternative when confronted with several alternatives that possess positive and negative qualities.

b. Description: Pugh selection is a method to evaluate several alternatives such as concepts, designs, processes, and models. The evaluation is a systematic analysis against each datum leading to selection of that alternative which best satisfies the assessment criteria. The method uses a matrix approach along with an iterative procedure to produce potential hybrids that may be better than any of the original ideas.

Pugh selection is an effective method for:

1. clarifying customer requirements
2. reviewing several alternatives against a "standard"; for example reliability model selection against a standard set of acceptance criteria
3. assuring that the selection criteria is understood by the project team
4. obtaining team consensus on the acceptability of specific attributes of an alternative
5. assuring that the best alternative is identified

The key steps in the Pugh selection method are as follows (one or multiple alternatives can be selected):

1. identify and describe all alternatives
2. list the criteria to be used to assess each alternative
3. select the current approach as the normative standard against which all alternative will be assessed as equal to, better than, or worse than based on each criteria
4. compare and score each alternative against the normative standard for each criteria; "s", "+", "-"
5. total the number of scores in each category for each alternative; eliminate or modify those alternatives with a significant number of "-"s.
6. create hybrid alternatives based on changing the "-"s to "+" or "s"s wherever possible
7. make the alternative with the highest "+" to "-" ration greater than 1.0 and no further "hybrid" alternative can be created, the optimal alternative

c. Benefits: The Pugh selection method can result in greater insight into requirements and solutions, a quick review of many alternatives and additional hybrid alternatives, better understanding of alternatives, a more objective selection process, and increased teamwork and communication among the team members. One direct application to the software reliability program would be in selecting appropriate software reliability prediction and/or estimation models based on specified criteria, such as found in reference [AIAAR013].

d. Case Evidence: Measures of best alternative selection.

e. Limitations: The Pugh selection method is subjective and provides mostly a qualitative judgement of alternatives. In addition it is not easy to differentiate between two closely matched alternatives.

f. References:

1. ANSI/AIAA R-013-1992, "AIAA Recommended Practice for Software Reliability," February 1993.

2. Mattson, C. A., and Messac, A., "Development of a Pareto-Based Concept Selection Method," 43rd AIAA/ASME/ASCE/AHS Structures, Structural Dynamics, and Materials Conference, Paper No. AIAA 2002-1231, Denver, CO, April 22-25, 2002.

3. Pugh, S., Creative Innovative Products Using Total Design, Addison-Wesley, Reading, MA, Editors: Clausing, D. and Andrade, R., 1990.

4. Ulrich, K., and Eppinger, S., Product Design and Development, McGraw-Hill, 2000, pp. 137-157.

C.1.11 Quality Function Deployment (QFD)

a. Purpose: Quality function deployment is intended to assure that customers requirements and expectations are met by incorporating the voice of the customer into product design and development.

b. Description: QFD is a structure, comprehensive planning process based on a series of matrices used to document, correlate, communicate and track the customer requirements throughout the organization and/or specific projects. The QFD planning matrix is called the "House of Quality" because of its shape. QFD may be used for many customer-related activities such as: business and market analysis to determine which products to develop; clarification of customer requirements; incremental design improvement on next generation designs; team focus on the critical and key priorities of the customer. In traditional QFD there are four phases in which the voice of the customer is deployed: (1) Product Planning - translating customer requirements into product characteristics; (2) Part Deployment - translating product characteristics into component characteristics; (3) Process Planning; and (4) Process Control. The steps for product planning, most important for determining customer requirements, include:

1. identify customer wants and needs - example: level of reliability;

2. rank the importance of the customer needs - example: reliability relative to ease of use;

3. define satisfaction measures for each need - example: design evidence, failure rate evidence from testing;

4. determine critical areas for project focus by mapping satisfaction measures to needs - example: matrix for requirements identification and ranking table traced to satisfaction measures;

5. establish design activities to meet the customer requirements - example: software FMECA, software FTA, software reliability prediction/estimation modeling, failure rate analysis.

c. Benefits: Greater understanding of the customer needs; interaction ensures better customer buy-in for the result; priorities are known by all on the project team; fewer and earlier changes in design; increased teamwork and better communication among team members; lower start-up cost.

d. Case Evidence: Identification of customer requirements, measures to known whether requirements are met, and design activities targeted to providing measurement evidence.

e. Limitations: Requires training to be most effective; scope of the effort may be extensive and require careful tailoring for most effective implementation; total effort may exceed capabilities of the team to provide.

f. References:

1. Akao, Y., Quality Function Deployment, Productivity Press, January 1990.

2. Akao, Y. and Mizuno, S., QFD: The Customer-Driven Approach to Quality Planning and Development, Productivity Press, April 1994.

3. Quality Function Deployment Institute, http://www.qfdi.org/

C.1.12 Reliability Allocation

a. Purpose: Distribute system reliability requirements to the components that make up that system.

b. Description: Reliability requirements are generally specified at the system level early in the life cycle. During architectural analysis system reliability requirements are allocated to individual system components, including hardware and software. It is usually necessary to perform trade-off studies to determine the optimum architecture which will meet reliability requirements. This may involve reassigning functionality between hardware and software components.

Reliability allocation for individual subsystems must ensure that hardware and software subsystem goals are well-balanced among themselves. Well-balanced usually refers to approximate relative equality of development time, difficulty, risk, or to the minimization of overall development cost.

Reliability allocation is closely coupled with reliability block diagrams (C.1.13), formal scenario analysis (C.1.5), and operational profiles (C.3.6) in the construction of a reliability model. Once the scenarios/profiles are understood, the reliability allocations are made in accordance with the critical mission segments. For computational units in which software executes, the allocation model will be serial between the computational hardware and computational software.

c. Benefits: If sufficient analysis is conducted to support the reliability allocation, the likelihood that reliability requirements will be met is greater. Also, it is more cost effective to analyze and allocate reliability requirements early in the life cycle than to wait until after a system is developed to find out that it doesn't meet reliability requirements.

d. Case Evidence: Early specification of the reliability measures expected for the software product(s) and the tradeoffs to balance cost, schedule, and risk against the reliability key performance parameter. Integrated system and software reliability modeling and planning.

e. Limitations: Mission scenarios and software operational profiles may not accurately model the operational use.

f. References:

1. Friedman, M. and Voas, J., Software Assessment: Reliability, Safety, and Testability, John Wiley & Sons, 1995.

2. Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

3. O'Connor, P., Practical Reliability Engineering, 3rd edition, John Wiley & Sons, 1991.

C.1.13 Reliability Block Diagrams

a. Purpose: Identify diagrammatically the set of events that must take place and the conditions which must be fulfilled for a system or task to execute correctly.

b. Description: Reliability block diagrams illustrate the relationship between system components with respect to the effect of component failures upon overall system reliability. These relationships generally fall into four categories: a serial system; a dual redundant system; m out of n redundant system; or a standby redundant system. An analysis of these relationships and their hazard rates will lead to an optimum system configuration which will meet specified reliability requirements. For a system composed of several components, the software element is integrated as a sequential (AND) component of each hardware computational component. The software may consist of one or more sequential subcomponents. Typically no more than a layering of communication software (usually commercial), operating system software (usually commercial), and application software (the target of main interest). The steps for integrating software reliability into the system reliability computation are as follows:

1. Divide system into block diagram components for reliability analysis;
2. Allocate system reliability objectives to individual components;
3. Calculate the component reliability from predicted subcomponent reliabilities - use software reliability in terms of calendar time;
4. Calculate system reliability from block component predicted reliabilities;
5. Select appropriate operating modes for reliability measurement;
6. Compare reliability component measurements against objectives;
7. Modify hardware/software until objectives are met, or change objectives.

The two basic paths on a reliability block diagram are the sequential (AND) path and the parallel (OR) path. For an AND path the reliability is simply the product of the reliabilities of the parts. For example:

$$R = R1*R2*R3*...*Rk \qquad \text{(Eq. C1)}$$

for sequential components 1,2,3, ... ,k. The block diagram and example computation for a computational hardware component that has a layer of commercial operating system and application software is illustrated below:

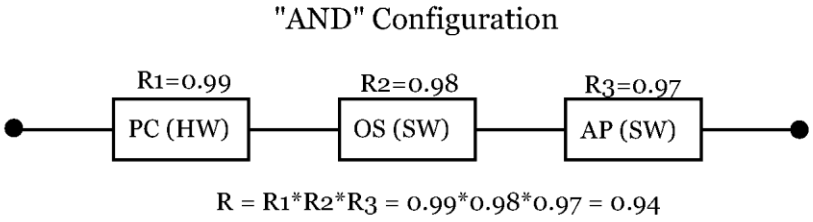

FIGURE C3—EXAMPLE SERIAL RELIABILITY BLOCK DIAGRAM COMPUTATION

For an OR of parallel paths the reliability is one minus the probability of failure of the whole. The probability of failure is the product of the probability of the failures of each component on a parallel path. For example:

$$R = 1-F = 1-(F1*F2*F3*...*Fk) = 1 - (1-R1)(1-R2)(1-R3)...(1-Rk) \quad \text{(Eq. C2)}$$

for parallel components 1,2,3,...,k. The block diagram for two parallel computational hardware components each of which has a layer of commercial operating system and application software is illustrated below:

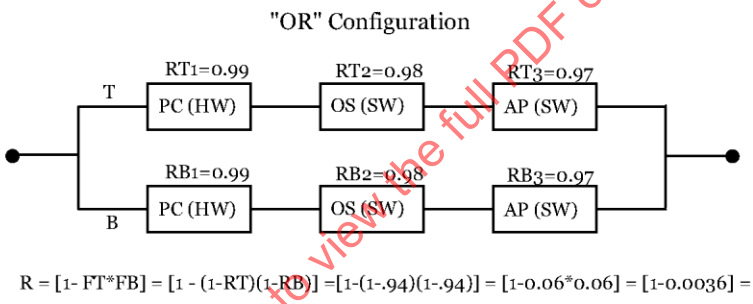

FIGURE C4—EXAMPLE PARALLEL RELIABILITY BLOCK DIAGRAM COMPUTATION

The combined computation component consists of a hardware component and a software component in serial relationship. That means that if either hardware or software fail, then the component fails. In addition, the software component is composed of two parts in series, the non-developmental software (e.g., OTS) and the newly developed application-specific software. If either of the two software parts fail, then the software fails, and consequently, the computational component fails. This model of a HW/SW element is illustrated in the figure below:

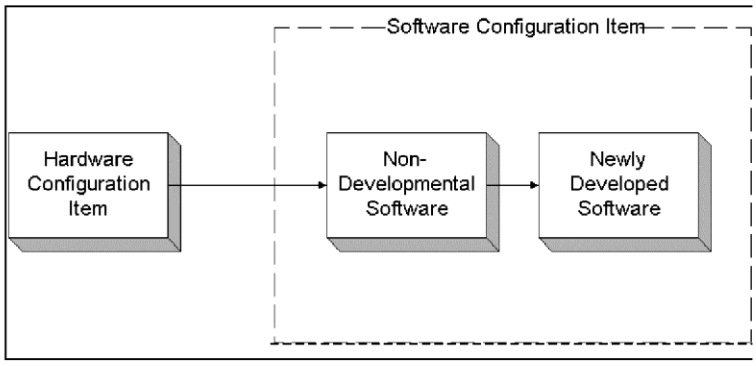

FIGURE C5—RELIABILITY BLOCK DIAGRAM MODEL OF HW/SW COMPUTATIONAL COMPONENT

c. Benefits: Reliability block diagrams are useful for analyzing systems which are composed of multiple diverse components, such as hardware and software.

d. Case Evidence: Identification of software component relationship to the system in terms of physical reliability computation.

e. Limitations: A reliability block diagram does not necessarily represent the system's operational logic or functional partitioning.

f. References:

1. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

2. IEC 61078:1991, "Analysis techniques for dependability - Reliability block diagram method," International Electrotechnical Commission, 1991.

3. Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

4. O'Connor, P., Practical Reliability Engineering, 3rd edition, John Wiley & Sons, 1991.

C.1.14 Reliability Prediction Modeling

a. Purpose: Predict the future reliability of software

b. Description: The failure probability of a new program, usually one that is under development, is predicted by comparing it to the known failure probability of an existing operational program. The criteria for determining the degree of similarity include: design similarity, similarity of service use profile, procurement and project similarity, and proof of reliability achievement. The generic process involves estimating the fault density per thousands of non-commented source lines of code (KNCSLOC) or function points. This value is then used to predict the number of errors remaining in the software and the time it will take to find them.

As an example of such a prediction model, the reference by Neufelder-Owner describes a model based on approximately 125 design and process parameters to predict delivered defect density. A process score "X" which is the sum of scores on the parameters is related to the defect density (defects per KNCSLOC, which could also be in terms of function points) as follows:

Process Score = X = sum of scores on 125 parameters (~1400 to 2800 range)

Predicted Delivered Defect Density per KNCSLOC(assembler) = D_a

$D_a = 0.00000017*X^2 – 0.00100439*X + 1.58463875$

$D_L = a* D_a$ = predicted delivered defect density in KNSLOC for language “L”; a = ~ 7 for C++

Converting to Reliability

N_0 = Inherent # delivered defects = $D_{C++}*4.5 = \sim 5.27$

Q = Ratio between N_0 and failures per time based on historical data/testing data

$\lambda(t) = N_0 * \exp(-Q*t/ N_0) / t = 5.27 * \exp (-Q*t/ N_0) / t$

$R(t) = \exp (- \lambda t)$

c. Benefits: The prediction model can be executed at any time during the software life cycle.

d. Case Evidence: Reliability prediction based on design parameters and other life cycle process information.

e. Limitations: The prediction validity depends on the similarity between the historical data upon which the prediction model is based and the target software program with its operational environment(s) and operational profile(s).

f. References:

1. ANSI/AIAA R-013-1992, “AIAA Recommended Practice for Software Reliability,” February 1993.

2. BS5760 Part 8: "Guide to the Assessment of Reliability of Systems Containing Software," British Standards Institution (BSI), October 1998.

3. Fenton, N., Software Metrics: A Rigorous Approach, Chapman & Hall, 1991.

4. IEEE Std. 982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE, 1988.

5. IEEE Std. 982.2-1988, "IEEE Guide for the Use of the Standard Dictionary of Measures to Produce Reliable Software," IEEE, 1988.

6. IEEE Std-1413-1998,"IEEE Standard Methodology for Reliability Prediction and Assessment for Electronic Systems and Equipment," IEEE Reliability Society, December 1998.

7. Lakey, Peter and Neufelder, Ann Marie, “System and Software Reliability Assurance Notebook,” Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

8. Lyu, M., Handbook of Software Reliability Engineering, McGraw-Hill, 1996.

9. Musa, J., Software Reliability Engineering, McGraw-Hill, 1999.

10. Neufelder-Owner, A., N., “The Facts About Predicting Software Defects and Reliability,” Journal of the RAC, 2ndQ, pp 1-4, 2002.

C.1.15 Response Time, Memory, Constraint Analysis

a. Purpose: To ensure that the operational system will meet all stated response time, memory and other constraints.

b. Description: Engineering analyses are conducted by an integrated product team to evaluate the system architecture and detailed design. The allocation of response time budgets between hardware, system software, and application software are examined to determine if they are realistic and comply with stated requirements. An assessment is made to determine if the available memory is sufficient for the system and application software. Minimum and maximum system throughput capacity under low, normal, peak, and overload conditions is estimated. Techniques such as Rate-Monotonic Analysis ensure the ability of real-time tasks to meet all critical deadlines even in the worst case system response time to events.

c. Benefits: Design deficiencies, hardware and software, are uncovered before full-scale development.

d. Case Evidence: Identification and removal of specific design deficiencies.

e. Limitations: This is a static analysis technique which should be supplemented by dynamic test methods (C.3.4) such as performance testing and stress testing.

f. References:

1. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

2. Sha, L., Klein, M.H., Goodenough, J., “Rate Monotonic Analysis for Real-Time Systems,” 129-155. Foundations of Real-Time Computing: Scheduling and Resource Management, Boston, MA: Kluwer Academic Publishers, 1991.

3. Klein, M.H., et al., A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems, Boston, MA: Kluwer Academic Publishers, 1993.

C.1.16 Six Sigma

a. Purpose: Quantify the design margins or product robustness.

b. Description: Six Sigma uses statistical tools to characterize how well the product can be expected to perform vis-à-vis its operating limits. Six Sigma is a quantitative process for problem resolution. It has been extended to Design for Six Sigma (DFSS) as a preventative design tool, to assure designs are 6 Sigma robust, meaning they will have fewer than 3.4 defects per million opportunities. Five key Six Sigma stages direct project execution: "Define, Measure, Analyze, Improve and Control" along with the associated terminology and tools. The corresponding project stages for Design for Six Sigma (DFSS) are: "Identify, Develop, Optimize, and Validate". The defined process steps, and their associated tools, promote proper project identification, development and closure.

Six Sigma has been greatly enabled by automated statistical tool sets such as Minitab and Statistical Analysis Software (SAS). Six Sigma graphical and statistical tools include: Process mapping, FMEA, Cause/effect Diagram, Descriptive Statistics, Hypothesis testing, Regression analyses, Relationship Mapping, Measurement Systems Analysis, Gage R&R, Analysis of Variance (ANOVA), Regression Fitting, Residual analyses, Process Capability (Cpk: before and after improvements), Z score (before and after improvements), Cost impact analyses, Histograms, Paretos, Trend analyses, Multivariate analyses, Correlation tests, Process map after fix, Simulations, Design of Experiments (DOE), Mistake Proofing, Control plans (goals, responsibilities, actions, metrics, schedules), Statistical Process Control (SPC) implementations, fault containment plans/systems, and feedback systems. A good overview and explanation of the Six Sigma tool set, and at what stage to apply the tools, can be found at http://www.isixsigma.com/library/content/c020617a.asp

c. Benefits: Six Sigma is a systematic, proven approach to problem solving. Management and engineering colleagues are more cooperative in supporting a project if it is identified as a Six Sigma project. Starting a Six Sigma project sets a good expectation. The tools promote "fact based decision" making over classical "intuitive" problem solving. It is the scientific approach.

d. Case Evidence: Statistical metrics related to delivered defects, defect rates, and development process.

e. Limitations: The success of the Six Sigma approach is predicated on top-down management support. It has failed when that is lacking. A company needs to culturally integrate Six Sigma into its culture to achieve the synergistic returns that are possible.

f. References:

1. Breyfogle, Forest, Implementing Six Sigma: Smarter Solutions Using Statistical Methods, John Wiley & Sons, New York, 1999.

2. Harry, Mikel, and Schroeder, Richard, Six Sigma, Doubleday, New York, 2000.

3. Pyzdek, Thomas, The Six Sigma Handbook, McGraw Hill, 2001.

C.1.17 Sneak Circuit Analysis

a. Purpose: Identify unintended or unexpected software logic paths or control sequences that could inhibit desired system functions or result in undesired system events.

b. Description: Sneak circuits are latent conditions that are inadvertently designed into a system which may cause it to perform contrary to specifications and affect reliability. The first step in sneak circuit analysis is to convert the source code into a topological network tree, identifying the patterns for each node of the network. The use and interrelationships of instructions are examined to identify potential sneak circuits. Categories of sneak circuits that are examined include: unintended outputs, incorrect timing, undesired actions, and misleading messages. The last step is to recommend appropriate corrective action to resolve any unintended anomalies discovered by the analysis.

c. Benefits: Latent defects are identified prior to a system being fielded.

d. Case Evidence: Defect prevention and removal data.

e. Limitations: Software sneak circuit analysis is somewhat labor intensive and as such should only be applied to critical system components.

f. References:

1. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

2. Raheja, D., Assurance Technologies: Principles and Practices, McGraw-Hill, 1991.

3. System Safety Society, System Safety Analysis Handbook, July 1993.

C.1.18 Software Failure Modes, Effects, and Criticality Analysis (SFMECA)

a. Purpose: Identify potential system and software failure modes, their effect and criticality, so that mitigating design features can be incorporated.

b. Description: A software FMECA consists of two important parts: identification through a system/hardware FMECA of software components that could potentially result in a system failure mode; and, application to the software of a procedure similar to a system/hardware FMECA for those identified system failure modes. Like a hardware FMECA, a software FMECA identifies design deficiencies and vulnerabilities which could affect reliability. This technique can and should be used iteratively throughout the software life cycle. The software is broken into logical components, such as functions or tasks. The potential software failure modes (particularly related to the identified system failure modes, but additional failure modes may be identified as well) are predicted for each component. Causes of these "potential" software failure modes and their effect on system behavior is postulated. Lastly, the severity and likelihood of each software failure mode is determined. The principle data elements to be collected and analyzed for each potential failure mode are: the failure, the cause(s), the effect of the failure, the criticality of the failure, the software component responsible, and the recommended mitigating control measure. The results of a software FMECA should be used to prioritize defect prevention and mitigation activities, in particular, design of the software so that the potential for such software failure modes is reduced.

Failure Criticality

Category	Severity	Description
1	System Abort	A software/firmware problem that results in a system abort or crash
2	System Degraded No Workaround	A software/firmware problem that severely degrades the system and no alternative work-around exists: restarts not acceptable
3	System Degraded Workaround	A software/firmware problem that severely degrades the system and an alternative work around exists: process can continue with more operator action: restarts not acceptable
4	System Not Degraded	A software/firmware problem that does not severely degrade the system or any essential function: restart acceptable
5	Minor	All other problems or non-functional faults

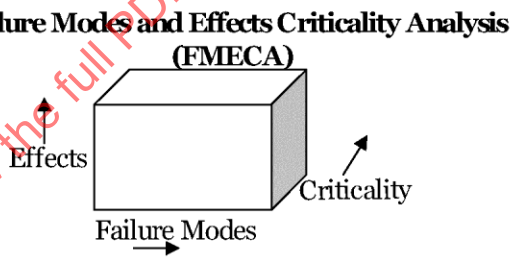

FIGURE C6—ELEMENTS OF A SOFTWARE FMECA

c. Benefits: A software FMECA can be merged with a system-level FMECA and input to a FTA. See SFTA (C.1.19). Potential for software failure modes is reduced through design and other mitigation strategies.

d. Case Evidence: Software failure modes, effects, criticality; defect prevention and mitigation strategies and results.

e. Limitations: The development of a SFMECA can be labor intensive unless an automated tool is used.

f. References:

1. AIR5080, "Integration of Probabilistic Methods into the Design Process," Society of Automotive Engineers, January 1997.

2. Herrmann, D., Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors, IEEE Computer Society, Los Alamitos, CA, 1999.

3. IEC 60812:1985, "Analysis Techniques for System Reliability – Procedure for Failure Modes Effects Analysis (FMEA)," International Electrotechnical Commission, 1985.

4. Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

5. Pries, K., "Failure Mode and Effects Analysis in Software Development," Society of Automotive Engineers, paper #982816, International Truck & Bus Meeting & Exposition, November 1998.

6. SAE J Standard 1739, " Potential Failure Mode and Effects Analysis in Design (Design FMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) and Effects Analysis for Machinery (Machinery FMEA)," Society of Automotive Engineers, August 2002.

7. Stamatis, D.H., Failure Mode and Effect Analysis: FMEA from Theory to Execution, ASQ Press, 1995.

8. System Safety Society, System Safety Analysis Handbook, July 1993.

C.1.19 Software Fault Tree Analysis (SFTA)

a. Purpose: Identify potential root cause(s) of undesired system events.

b. Description: A software FTA follows the same procedure as a hardware or system FTA to identify the root cause(s) of a major undesired event. SFTA is an extension of the SFMECA (C.1.18) activity in that the identified potential software failure modes are analyzed in terms of what potential software faults (single point of failure) or multiple faults (multiple points of failure) might result in the potential software failure mode. SFTA is particularly useful in the analysis of events, or combinations of events, that will lead to a hazard. Starting at an event which would be the immediate cause of a hazard, the analysis is carried out backward along a path. Combinations of causes are described with logical operators (AND, OR, IOR, EOR). Intermediate causes are analyzed in the same way back to the root cause. A software FTA should be developed iteratively throughout the software life cycle and in conjunction with a SFMECA (C1.18).

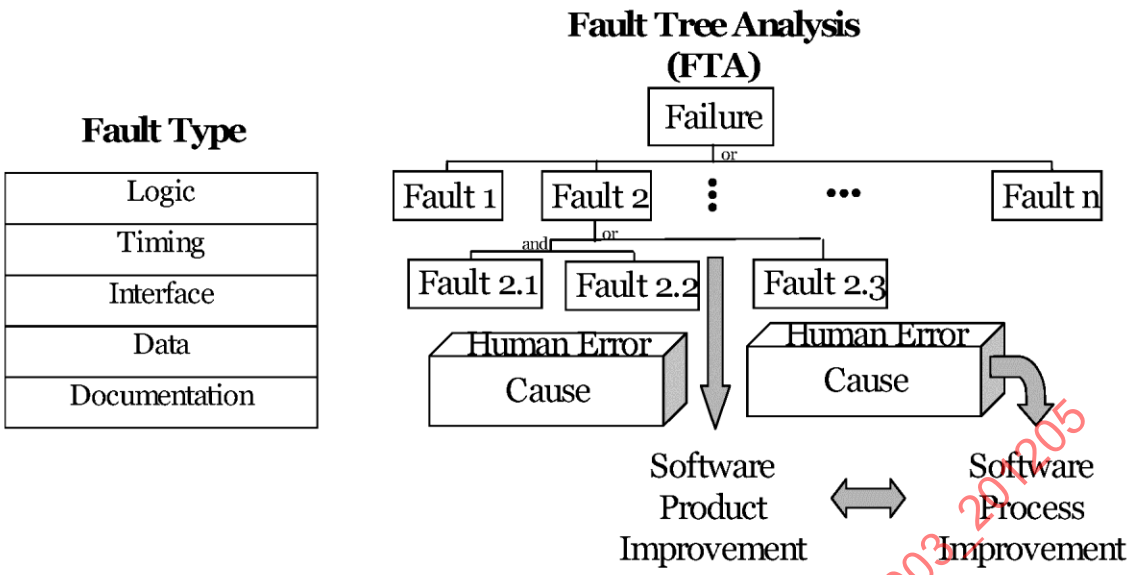

FIGURE C7—ELEMENTS OF A SOFTWARE FTA

c. Benefits: A software FTA can be merged into a hardware or system-level FTA. Identified software failure mode root causes can be mitigated by using the SFTA process early in the development/support process through design modifications.

d. Case Evidence: Potential failure root causes, mitigation strategy, verification results of design modifications.

e. Limitations: The development of a SFTA can be labor intensive unless an automated tool is used.

f. References:

1. IEC 61025:1990, "Fault Tree Analysis (FTA)," International Electrotechnical Commission, 1990.

2. Lakey, Peter and Neufelder, Ann Marie, “System and Software Reliability Assurance Notebook,” Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

3. Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

4. System Safety Society, System Safety Analysis Handbook, July 1993.

C.1.20 Software Reliability Engineering (SRE-Musa)

a. Purpose: SRE is a practice for *quantitatively* planning and guiding software development and test, with emphasis on reliability and availability.

b. Description: Although the term "Software Reliability Engineering" is used in a more general sense throughout this document, the SRE method described here is a specific instance of how one might conduct a software reliability engineering program. This particular SRE approach works by quantitatively characterizing and applying two things about the product: the expected relative use of its functions and its required major quality characteristics. The major quality characteristics are reliability, availability, delivery date, and life-cycle cost. In applying SRE, you can vary the relative emphasis you place on these factors. Underlying the methodology is the basic premise of testing software under operational conditions and use of reliability estimation (C.3.9) growth models. In discussing SRE, you should always be thinking of total systems that also contain hardware and often human components.

When you have characterized use, you can substantially increase development efficiency by focusing resources on functions in proportion to use and criticality. You also maximize test effectiveness by making test highly representative of use in the field. You choose software reliability strategies to meet the objectives, based on data collected from previous projects. You also track reliability in system test against its objective to adjust your test process and to determine when tests may be terminated.

c. Benefits: SRE is a proven, standard, widespread best practice. SRE has been an AT&T Best Current Practice since May 1991 (Lyu 1996). SRE is also a standard practice. McGraw-Hill published an SRE handbook in 1996 (Lyu 1996). SRE is a widespread practice. There have been over 65 published articles by users of SRE, and the number continues to grow. SRE is widely applicable. From a technical viewpoint, you can apply SRE to any software-based product, starting at the beginning of any release cycle. From an economic viewpoint, you can apply SRE to any software-based product also, except for very small components, perhaps those involving a total effort of less than 2 staff months. SRE is independent of development technology and platform. It requires no changes in architecture, design, or code, but it may suggest changes that would be useful. It can be deployed in one step or in stages. SRE is very customer-oriented: it involves frequent direct close interaction with customers. It is highly correlated with attaining Levels 4 and 5 of the Software Engineering Institute Capability Maturity Model. Despite the word "software," software reliability engineering deals with the entire product, although it focuses on the software part. It takes a full-life-cycle, proactive view, as it is dependent on activities throughout the life cycle. It involves system engineers, system architects, developers, users (or their representatives, such as field support engineers and marketing personnel), and managers in a collaborative relationship. The cost of implementing SRE is small. There is an investment cost of not more than 3 equivalent staff days per person in an organization. The operating cost over the project life cycle typically varies from 0.1 to 3 % of total project cost.

d. Case Evidence: Quantitative measures of estimated software reliability through systematic software reliability engineering approach; predicted operational reliability; estimated corrective maintenance actions.

e. Limitations: Not significant, as demonstrated by more than 65 published articles by users who have successfully employed the practice. Accuracy depends on operational profile specification, model parameter estimates, and sufficient test data.

f. References:

1. ANSI/AIAA R-013-1992, "AIAA Recommended Practice for Software Reliability," February 1993.

2. Lyu, M., Handbook of Software Reliability Engineering, McGraw-Hill, New York, 1996.

3. Musa, J. D., Software Reliability Engineering: More Reliable Software, Faster Development and Testing, ISBN 0-07-913271-5, McGraw-Hill, New York, 1998.

4. Musa, J. D., "More Reliable Software Faster and Cheaper (Software Reliability Engineering)," 2002. website(updated regularly).: http://members.aol.com/JohnDMusa/

C.1.21 Statistical Analysis

a. Purpose: Provide standard techniques for collecting data and transforming it into information that can improve decision making under data variability and uncertainty.

b. Description: Statistical analysis provides the basis for many of the other reliability techniques and methods such as design of experiment (C.1.4), probabilistic methods (C.1.9), reliability prediction modeling (C.1.14), six sigma (C.1.16), software reliability engineering (C.1.20), and reliability estimation modeling (C.3.9). Decisions are based on many different standard statistical techniques using general features of data such as trend over time, clustering of data points, evidence of non-randomness, and presence of 'outliers'. The concepts of population/sample, point estimation, confidence interval estimation, hypothesis testing, and probability/probability distributions are fundamental to statistical analysis.

Descriptive statistics deals with organization, summarization, and presentation of data for populations or samples. Inferential statistics addresses use of the information in a population sample to draw conclusions about the population. Probability builds the base that enables one to understand how inference and decision-making techniques are developed, why they work, and how the conclusions from these inferential techniques can be interpreted and presented correctly. Probability provides the mathematical foundation and language of inferential statistics that helps us understand the variability (known information) of data and the uncertainty (unknown information) about data.

Reliability models and categorical classifications have been developed based on statistical analysis. For example, one classification of such models is as prediction models (C.1.14) or estimation models (C.3.9). Both categories of models use statistical analysis of data to justify the initial model definition as well as to justify whether such a model is appropriate to use in a given application. Another classification is in terms of stochastic reliability models as General (e.g., graphical, time-series analysis), Black Box (e.g., fault activation, failure trend, environmental), and Structural (e.g., modular, hierarchical, hardware/software). This classification scheme is described in the reference [2] below.

Some examples of useful estimates that can be provided by certain classes of stochastic reliability models include: probability that software will not fail in a given period of operation; mean time to next activation of a new software fault; current failure rate; predicted failure rate after a given further period of trial and fault correction; expected number of faults that will be activated in a given further period of operation; further time required for trial and fault correction in order to achieve a defined target value of any of the previous items.

c. Benefits: Analyses can provide features of data that are ignored by black-box parametric models; anomalies in data that may violate parametric models can be detected; non-parametric models graphical analyses are simple to apply and an abundance of tool support exists.

d. Case Evidence: Statistical measures and confidence evidence in those measures.

e. Limitations: Techniques do not generally model the mechanism of failure, hence long term predictions of future reliability growth may not be applicable; data anomalies can be detected but the cause must be determined by some other method such as root cause analysis (C.3.10).

f. References:

1. ANSI/AIAA R-013-1992, “AIAA Recommended Practice for Software Reliability,” February 1993.

2. BS5760 Part 8: "Guide to the Assessment of Reliability of Systems Containing Software," British Standards Institution (BSI), October 1998.

3. Hines, W.W., and Montgomery, D.C., Probability and Statistics in Engineering and Management Science, Third Edition, John Wiley & Sons, New York, NY, 1990.

C.2 Design Techniques

C.2.1 Design by Contract

a. Purpose: Define a methodology through which software components can be constructed correctly through specifications (or contracts) that govern the interaction of the component with any potential user.

b. Description: The notion of Design by Contract is central in the systematic approach to object-oriented software construction, as embodied in the Eiffel method. Reliability, although desirable in software construction regardless of the approach, is particularly important in the object-oriented method because of the special role given by the method to reusability: unless we can obtain reusable software components whose correctness we can trust much more than we trust the correctness of usual run-of-the-mill software, reusability is a losing proposition. Under the Design by Contract theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations -- contracts. The Design by Contract suggests associating a specification with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world. A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope. The same ideas apply to software. This contract governs the relations between the routine and any potential caller. It contains the most important contract information about the routine: what each party must guarantee for a correct call, and what each party is entitled to in return.

The principle constructs of the software component contract include:

1. precondition: requirements that must be satisfied when a routine is called;
2. postcondition: requirements that must be satisfied when a routine ends;
3. class invariants: requirements all class routine pre/post conditions must satisfy;
4. exception handling: raising exception to caller or retrying an alternative; and
5. other constructs: check of preconditions by client; retention of original inputs for output comparison.

Providing an environment, including a specific language, in which these elements are naturally implemented is an integral part of the Design by Contract approach. The Eiffel method includes an implementation of the Design by Contract approach. In addition, there are limited forms of the Design by Contract constructs available for several languages, although these elements must be artificially designed, typically in the form of assertions. In the Eiffel language, the constructs are directly implemented.

c. Benefits: The benefits of Design by Contract include: (1) A better understanding of the object-oriented method and, more generally, of software construction.; (2) A systematic approach to building bug-free object-oriented systems; (3) An effective framework for debugging, testing and, more generally, quality assurance; (4) A method for documenting software components; (5) Better understanding and control of the inheritance mechanism; (6) A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

d. Case Evidence: Requirements traceability to contract level; precise test coverage/measures;

e. Limitations: Requirements are not always able to be precisely described in terms of low level constructs, although attempting to do so will still facilitate the requirement specification and tracing process.

f. References:

1. Meyer, Bertrand, Object-Oriented Software Construction, Prentice-Hall, 1997 (2nd Edition).

2. Meyer, Bertrand, "Applying Design by Contract," IEEE Computer, Col. 25, No. 10, Oct 1992, pp 40-51.

3. Meyer, Bertrand, Eiffel: The Language, Prentice-Hall, 1992.

4. Home page for the Eiffel products: http://www.eiffel.com/

C.2.2 Fault Tolerant Design

a. Purpose: Detect and respond to erroneous system states in order to contain the results.

b. Description: There are three categories of fault tolerance: system , hardware, and software. Hardware fault tolerance is usually implemented through redundancy. The concept is that if a primary hardware component fails, the redundant component will take over and continue normal operations. Software fault tolerance is usually implemented through block recovery, degraded mode operations, diversity, error detection/correction algorithms and other design techniques. Because it is nearly impossible to develop software that is 100% free of defects, design techniques should be employed to detect and recover from errors while minimizing the consequences of those errors. System fault tolerance combines hardware and software fault tolerance, with software monitoring the health of both the hardware and software. System fault tolerance should be employed for high integrity, mission critical systems; especially those that include embedded real-time applications.

Block recovery refers to design features that provide correct functional operation in the presence of one or more errors. Backward block recovery resets a system to a previous known safe state if an error is detected. Forward block recovery forces a system to a future known safe state if an error is detected.

Degraded mode operations provides an intermediate state between full operation and system shut-down. This allows minimum system functionality to be maintained until corrective action can be taken. During the design and development of high integrity, mission critical systems a minimum required set of functionality should be identified, along with the conditions under which the system should transition to this mode. Design features should ensure this functionality set will be operational in the presence of one or more failures. A maximum time interval for a system to remain in degraded mode operations should be defined. A method of notifying operational staff that a system has transitioned to degraded mode operations should be identified.

Error detection and correction algorithms have been used to increase the reliability of service provided. Such algorithms expand upon returning an error code by automatically responding to the error condition. These techniques are equally applicable to other high integrity, mission critical systems; particularly those that employ embedded real-time systems. The technique involves: (1) identifying where possible errors could occur in accessing, manipulating, and relaying information; (2) identifying inconsistent program states, flows or timing, and (3) defining the appropriate corrective action to be taken in each instance. The priority for corrective action is determined by correlating the severity of the consequences and likelihood of occurrence of error conditions and states to specified reliability requirements.

Diversity refers to using different means to perform a required function or solve the same problem. Diversity limits the potential for common cause failures. For software, this means developing more than one algorithm to implement a solution. The results from each algorithm are compared and if they agree, the appropriate action is taken. Depending on the criticality of the system, 100% agreement or majority agreement may be implemented. If the results do not agree, error detection and recovery algorithms take control.

c. Benefits: Fault tolerant design can increase the reliability of critical system functions and components.

d. Case Evidence: Fault tolerant design architecture and test results verifying fault tolerance implementation.

e. Limitations: Fault tolerance increases the physical size and complexity of a system through hardware redundancy, software diversity and other design features. This may conflict with specified size constraints.

f. References:

1. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

2. Levi, Shem-Tov, Fault Tolerant System Design, McGraw-Hill, 1994.

3. Lyu, M. (ed.), Software Fault Tolerance, John Wiley & Sons, 1995.

C.2.3 Formal Methods/Languages

a. Purpose: Provide a mathematically based technique for describing system properties within which systems can be specified, developed, and verified in a systematic rather than ad hoc manner.

b. Description: Formal methods involve the use of logically precise specifications based on discrete mathematics. This type of mathematics is well suited for modeling discrete systems, especially those involving logical interactions. These formal specifications greatly facilitate the modeling of requirements and high level design. Typically a formal language provides the syntax and semantics for representing the specifications.

The primary types of analysis supported by formal methods are checking the internal consistency of a specification and proving that the system specified satisfies desired properties. These types of analyses can be partially automated using computer-based tools that not only support the initial development and analysis of specifications, but also reduce the time required for re-analysis in response to subsequent modifications or extensions. This type of rigorous analysis is very useful for smaller, critical components that must satisfy rule-based requirements such as required to support safety and/or security functions.

c. Benefits: Formal methods enable defects in requirements to be detected earlier than otherwise, and can reduce the number of mistakes in interpreting, formalizing, and implementing correct requirements. Formal methods define formalized statements that can be analyzed and more rigorously verified. Formal methods cause more defects to be detected than would otherwise be the case and provide mechanisms that support the objective to guarantee the absence of certain defects. Formal methods can provide compelling evidence of correctness early enough to be useful, cheaply enough to be feasible, and on the basis of modeling that is simple enough to be credible.

d. Case Evidence: Formal specifications, proofs of correct specifications, potential reduced defect delivery.

e. Limitations: Currently formal methods can only be applied to small, critical system components not to large, complex systems. Such methods require expertise beyond typical software engineers. Formal methods do not guarantee a superior product. As with all tools, the potential benefits of formal methods can be realized only if the tools are judiciously applied to suitable applications. Formal methods may provide less benefit than anticipated due to anomalies such as erroneous specifications or flawed verifications.

f. References:

1. NASA, "Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning And Technology Insertion," NASA/TP-98-208193, Release 2.0, Washington DC, December 1998.

2. NASA, "Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume II: A Practitioner's Companion," NASA-GB-001-97, Washington DC, December 1997.

3. NASA Formal Methods Web Site, http://shemesh.larc.nasa.gov/fm/

4. Sheppard, D., An Introduction to Formal Specification with Z and VDM, McGraw-Hill, 1995.

5. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw Hill, 1997 (4th Edition).

C.2.4 Independence, Isolation, Inoperability, Incompatibility (I4)

a. Purpose: Provide principles of design and implementation that facilitate effective testing, safety/security-critical function implementation, and localization of the effects of failure.

b. Description: The I4 principles are well-known within the safety and security communities. For example, an operating system safety/security kernel with clear separation of software safety/security functions from all other software functions provides a less complex target for application of safety-related methods such as software FMECA (C.1.18), software FTA (C.1.19), and formal methods (C.2.3).

Independence - multiple, independent subsystems and completely different sources of enabling stimuli for critical functions are incorporated within the system. For software, this might mean multiple independence conditions (e.g., multiple hardware unique signals) must occur before the software safety function is activated.

Isolation - critical functions are encapsulated separate from any other functions that might cause undefined interactions with the critical functions. For software, information hiding and modularity are common methods for isolating functionality. Architecture designs emphasize identification and functional isolation of the critical functions. Safety and security functions are typical examples of critical functions.

Inoperability - critical functions become predictably and irreversibly inoperable in credible abnormal operating environments before the isolation features are compromised. For software, this might be a fault tolerant implementation in combination with hardware. The system in an abnormal environment would degrade to an inoperable state prior to a safety or security breech, or transition to a safe state.

Incompatibility - functional interfaces are constructed so that they are incompatible with functions (in particular safety critical functions) with which they are not intended to interface. For software, this might be implemented through a design by contract (C.2.1) interface control specification mechanism.

Information hiding or encapsulation is a design technique that minimizes the interdependency or coupling of modules and maximizes the independence or cohesion of modules. The interface to each software module is designed to reveal as little as possible about the module's inner workings and provide appropriate responses to incompatible interface requests. This is accomplished by making the logic of each module and the data it utilizes as self contained as possible. The likelihood of common cause failures is reduced, fault propagation is minimized, and future maintenance and enhancements are facilitated.

c. Benefits: Facilitates reliability claims and evidence of successful implementation; provides a consistent framework within which case evidence can be presented.

d. Case Evidence: Organizing framework in which reliability/safety/security claims and evidence can be negotiated with and presented to the customer and/or certification authority.

e. Limitations: The definitions of evidence supporting the I4 principles is not always precise and requires considerable negotiation among customer, supplier, and in some cases the certification authority.

f. References:

1. Harrison, R., Counseli, S., and Nithi, R., “An Evaluation of the MOOD Set of Object-Oriented Software Metrics,” IEEE Transactions on Software Engineering, Vol. 24, No. 6, June 1998, pp 491-496.

2. Leveson, N. G., Safeware: System Safety and Computers, Addison Wesley Publishing Company, 1995.

3. Mazzanti, F., "Coding Regulations for Safety Critical Software Development," Proceedings of the 2nd IEEE Software Engineering Standards Symposium (ISESS’95), 1995.

4. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December 1972, pp. 1053-1058.

C.2.5 Mistake/Error Proofing

a. Purpose: Prevent the special causes that result in defects, or inexpensively inspect each item that is produced to determine whether it is acceptable or defective. Error proofing typically pertains to product design and mistake proofing to eliminating potential implementation/production/manufacturing related mistakes.

b. Description: Shigeo Shingo was one of the industrial engineers at Toyota who has been credited with creating and formalizing Zero Quality Control (ZQC), an approach to quality management that relies heavily on the use of mistake-proofing (called poka-yoke in Japanese) devices. These devices are used either to prevent the special causes that result in defects, or to inexpensively inspect each item that is produced to determine whether it is acceptable or defective. A mistake-proofing device is any mechanism that either prevents a mistake from being made or makes the mistake obvious at a glance.

Shingo identified three different types of inspection: judgment inspection, informative inspection, and source inspection. Judgment inspection involves sorting the defects out of the acceptable product, sometimes referred to as "inspecting in quality." Informative inspection uses data gained from inspection to control the process and prevent defects. Traditional Statistical Process Control (SPC) is a type of informative inspection as are formal in-process reviews (C.3.5). Both successive checks and self-checks are also a type of informative inspection. Source inspection determines "before the fact" whether the conditions necessary for high quality production exist. With source inspection, mistake-proofing devices ensure that proper operating conditions exist prior to actual execution/production. Often these devices are also designed to prevent execution/production from occurring until the necessary conditions are satisfied. This type of device is known as a "forcing function." Source inspection is intended to keep defects from occurring. Self-checks and successive checks provide feedback about the outcomes of the process. Self-checks and successive checks should be used when source inspection cannot be done or when the process is not yet well enough understood to develop source inspection techniques.

One specific "error proofing" software technique is "defensive programming", a software design technique in which critical system parameters and requests to transition system states are verified through multiple diverse means before acting upon them. User interface design, fault tolerance checks, and standard programming practices result in a more robust system architecture and immunity from (some) transient faults.

c. Benefits: Low cost defect prevention and removal devices that promote good quality practices by the persons performing the activity.

d. Case Evidence: Identified life cycle activities that contribute to reduced software defects/faults.

e. Limitations: Complexity of the activities may prevent mistake-proofing devices from being identified and/or used effectively.

f. References:

1. Bayer, P.C., "Using Poka Yoke (Mistake Proofing Devices) to Ensure Quality," IEEE 9th Applied Power Electronics Conference Proceedings 1:201-204, 1994.

2. Chase, R.B. and Stewart, D. M., Mistake-proofing: Designing Errors Out, Productivity Press, Portland, Oregon, 1995.

3. Grout, J.R., "Mistake-Proofing: Process Improvement Through Innovative Inspection Techniques," The Quality Yearbook: 405-414, 1998.

4. Marchwinski, Chet (ed.), "Microsoft, HP Use Poka-Yoke to Squash Software Bugs," Productivity 18(5): 1-4, 1997.

5. Shingo, S., Zero Quality Control: Source Inspection and the Poka-Yoke System, Productivity Press, Cambridge, MA, 1986.

C.2.6 Petri Nets

a. Purpose: Identify potential race and/or deadlock conditions, particularly in real-time applications.

b. Description: Petri nets are used to model relevant aspects of system behavior at a wide range of abstract levels. Petri nets are a class of graph theory models which represent information and control flow in systems that exhibit concurrency and asynchronous behavior. A Petri net is a network of states and transitions. The states may be marked or unmarked; a transition is enabled when all the inputs places to it are marked. When enabled, it is permitted but not obliged to fire. If it fires, the input marks are removed and each output place from the transition is marked instead. These models can be defined in purely mathematical terms, which facilitates automated analysis such as producing reachability graphs.

A Petri Net fragment derived from the example found in the reference [5] of a skidding, slipping, steering automobile design with Anti Lock Brakes is illustrated below. The nomenclature is fairly obvious, for example, for an abnormal steering the "system" detects this and begins to make automated braking (right front, left front, right rear, left rear – or combinations) adjustments depending on what the abnormal steering is. These "state event" and "transitions" are illustrated in the Petri net graph. The fail states depend on the motion physics of the automobile and can be tied to a reliability model to predict the reliability average time prior to the system failure. The availability models can also predict the percentage of time the system will be operational. Automobile design can then be appropriately adjusted to balance in an optimum way both the reliability and availability – within the aesthetic constraints given to the design engineer.

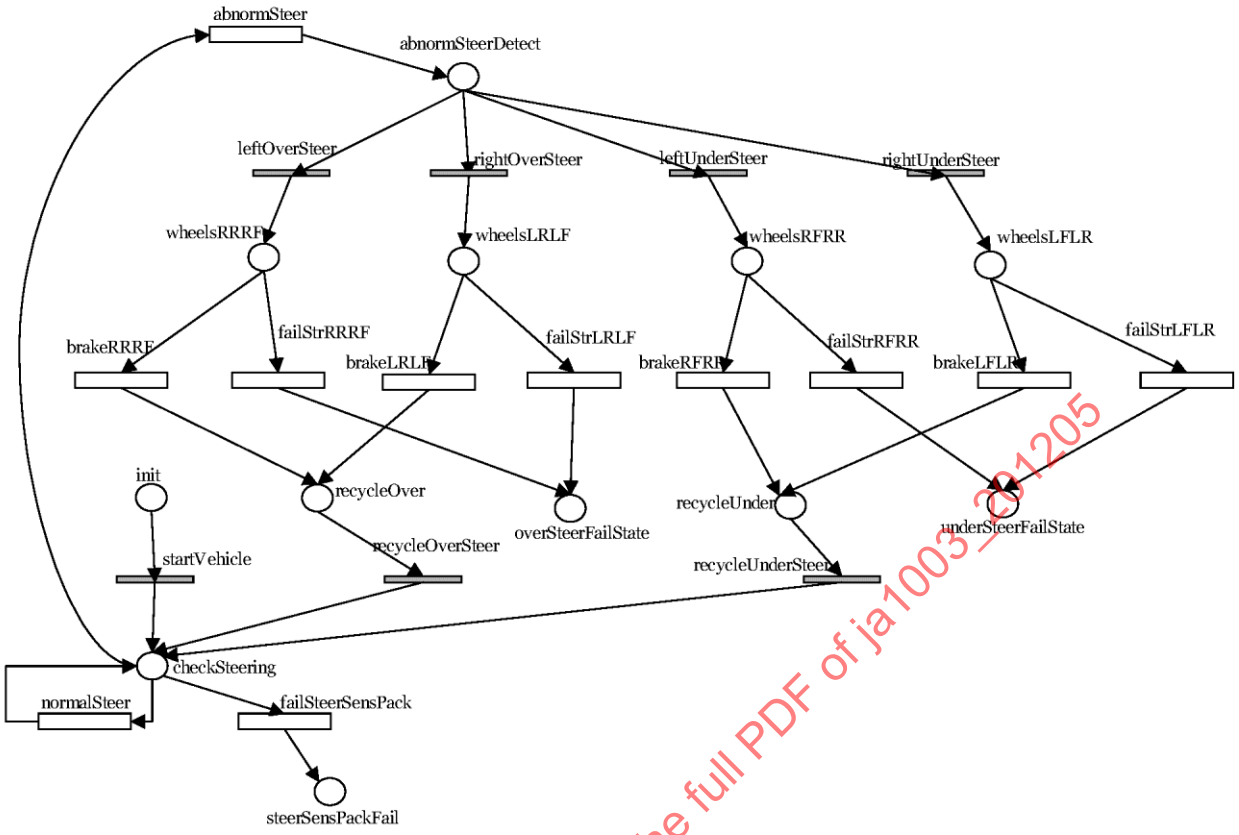

FIGURE C8—EXAMPLE SOFTWARE PETRI NET DIAGRAM (FRAGMENT)

c. Benefits: Petri nets can be used to model an entire system, subsystems, and/or subcomponents at conceptual, top level design, and implementation levels. They are useful for identifying deadlock, race, and nondeterministic conditions. Particularly applicable for systems/software that is safety critical.

d. Case Evidence: Identification of state conditions that could cause failures; mitigation through design evidence and formal analysis that such failures can not occur;

e. Limitations: The production of Petri nets can be time consuming without the use of an automated tool.

f. References:

1. Buy, U. and Sloan, R. H., "Analysis of Real-Time Programs with Simple Time Petri Nets," Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA), ACM Press, p. 228-239.

2. Jensen, K. "Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use," Springer-Verlag, vol. 1, 1996, vol. 2, 1995.

3. Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

4. Peterson, J. L., Petri-Net Theory and the Modeling of Systems, Prentice Hall, 1981.

5. Sheldon, F., Greiner, S., and Benzinger, M., “Specification, Safety and Reliability Analysis Using Stochastic Petri Net Models,” Proceedings of the Tenth International Workshop on Software Specification and Design, IEEE, November, 2000.

6. System Safety Society, System Safety Analysis Handbook, July 1993.

C.2.7 Software Integrity Checks

a. Purpose: Identify and prevent failure scenarios where the cause for the failure is either (accidental or malicious) corruption of the software itself, or environmental aspects have accidentally changed so that the software executes under non-designed for conditions.

b. Description: These design techniques ensure that the code to be executed and the data to be processed have not been altered and are of the correct type. Basically, these techniques can be divided into the following: (a) checksum checks, (b) configuration checks and (c) run-time state integrity checks.

Checksum tests are basically used to guarantee that the code or data has not been modified, by simply treating the code as a sequence of integers, adding them up, and checking them against a value stored within the code or data itself. These checks serve also as a safeguard against memory failure or corruption, or virus infection, and are typically performed upon the start of the software. During operational tasks such as software loading, this may be further reinforced by incorporating into the loading/communication protocols mechanisms such as Hamming codes (for self-correction) and Cyclic Redundancy Checks (CRCs), in order to prevent loading of corrupt code or data. The files to be loaded (code or data) may also have such mechanisms to ensure their integrity prior to loading. For example, the use of digital signatures applied to software code or data provides a higher integrity form of “checksum” test.

Configuration checks are particularly useful in high integrity systems, and consist of software that checks its own components (e.g., independent libraries) for availability and compatibility. Component signature exchanges allows confirmation that components are intact and not corrupted are compatible with the software wishing to use them. Another configuration check can be performed with the data to be processed (mission file, or database), if the data can inform about its structural version, so software can detect formats it is not able to process or older versions for which it can adapt. These mechanisms are also a safeguard against incomplete or incorrect installations and stop most attempts to install unauthorized components that may compromise a secure system. Configuration checks are made usually “on first use” of a specific component or data file/database.

Run-time state integrity checks are typically used in hard real-time systems, particularly in the case of multitasking applications where it is difficult to predict the exact system behavior at a given moment. Though similar to the defensive programming described in (C.2.5), no transition system states are verified, but only a "snapshot" of the current state is taken, to ensure that it is one of the acceptable ones. These checks may have a memory, for example, to ensure that no subsequent checks share the same state (which may indicate a deadlock). A characteristic of these run-time state integrity checks is that a) they are periodic during the whole execution b) they are very simple and c) are usually carried out by an external, interrupt-driven, program.

c. Benefits: Provide additional robustness against unexpected (and non-designed for) anomalous behavior, lack of operational configuration, and accidental or malicious corruption of code or data. Checks are very local & do not impact the overall code.

d. Case Evidence: Results of testing corrupted code, incorrect data formats, or deadlock states.

e. Limitations: Impacts on component reusability, "block upgrades" required for software updates, somewhat more complex design.

f. References:

1. ARINC Report 665-1, "Loadable Software Standards", Aeronautical Radio, Inc., January 2001.

2. ARINC Report 667, "Guidance for the Management of Field-Loadable Software (FLS)", Aeronautical Radio, Inc., May 2002.

3. ARINC Report 619-4, Supp. 4, "Airborne Computer High Speed Data Loader", Aeronautical Radio, Inc., May 2002.

4. AEEC Project Paper 666, "Electronic Distribution of Software", May 2002.

5. JA1005, SAE Surface Vehicle/Aerospace (JA) Standard 1005, "Software Supportability Program Implementation Guidelines," Society of Automotive Engineers, 2001.

6. RCTA/DO-178B/ED-12B, "Software Considerations in Airborne Systems and Equipment," Federal Aviation Administration software standard, RTCA Inc., December 1992.

C.3 Verification Techniques

C.3.1 Boundary Value Analysis

a. Purpose: Identify software errors when processing at specified parameter limits.

b. Description: During boundary value analysis, test cases are designed which exercise the software's parameter processing algorithms. Specific situations which are evaluated include:

- parameter below minimum specified threshold,
- parameter at minimum specified threshold,
- parameter at maximum specified threshold,
- parameter over maximum specified threshold, and
- parameter within specified min/max range.

The intent is to verify that the software responds to all parameters correctly: triggering error handling routines if a parameter is out of the specified range or following normal processing if a parameter is within the specified range. Boundary value analysis can also be used to verify that the correct data type is being used: alphabetic, numeric, integer, real, signed, pointer, and so forth.

c. Benefits: Boundary value analysis enhances system integrity by ensuring that data is within the specified valid range before operating upon it.

d. Case Evidence: Coverage measures for boundary values; testing results for boundary conditions.

e. Limitations: May be complex and time-consuming for testing if there are multiple parameters and boundary conditions and possible combinatorial constraints related to how the parameter values can be integrated.

f. References:

1. IEC 61508-7:1998-09-15, " Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

C.3.2 Cleanroom

a. Purpose: Prevent defects from being introduced and/or remaining undetected throughout life cycle activities.

b. Description: Cleanroom is specialized version of the incremental software model that uses formal methods and a statistical analysis techniques to support the measurement and analysis of pre-release software reliability. Cleanroom analysis emphasizes the prevention of errors, rather than just their detection and makes extensive use of formal methods and proofs. This technique takes a holistic view of software development by promoting iterative, top-down stepwise refinement of the total design, with correctness and verification of that design required at each step. The Cleanroom process emphasizes rigor in specification and design, and formal verification of each element of the resultant design model using correctness proofs that are mathematically based. Aspects of formal methods (C.2.3) may be part of the Cleanroom process. The Cleanroom approach also emphasizes techniques for statistical quality control, including testing that is based on the anticipated use of the software by customers. The Cleanroom Reference Model is expressed in terms of a set of 14 Cleanroom processes and 20 work products. It is intended as a guide for Cleanroom project management and performance, process assessment and improvement, and technology transfer and adoption.

c. Benefits: Cleanroom promotes error prevention and early detection of errors, when it is easier and cheaper to fix them.

d. Case Evidence: Rigorous design activities and measures of defect prevention through techniques such as defect removal efficiency (C.1.3).

e. Limitations: Cleanroom analysis does not determine if performance and response time requirements will be met.

f. References:

1. Dyer, M., The Cleanroom Approach to Quality Software Development, John Wiley & Sons, 1992.

2. Linger, R.C. and Trammell, C.J., “Cleanroom Software Engineering Reference Model, Version 1.0,” CMU/SEI-96-TR-022, Software Engineering Institute, Carnegie Mellon University, November 1996.

3. Mills, H.D., Dyer, M., and Linger, R., "Cleanroom Software Engineering," IEEE Software, September 1987, pp 19-24.

4. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw Hill, 1997 (4th Edition).

5. Prowell, S.J., Trammell, C., Linger, R., and Poore, J.H., Cleanroom Software Engineering - Technology and Process, Addison-Wesley, 1999.

C.3.3 Coverage Analysis

a. Purpose: Identify which parts of the software product have been tested by a specified set of test cases; identify an appropriate set of software test cases needed to meet test coverage goals.

b. Description: During test planning, the set of all possible test cases is examined to determine which test cases are redundant in that they test the same functionality and/or logic path. The intent is to obtain the highest possible test coverage with the least possible number of test cases. A thorough understanding of the system design and its functionality are needed construct an effective test architecture. There are several measures of test coverage for software, such as:

1. requirements coverage: percentage of requirements verified/validated by the specified test set;

2. feature coverage: percentage of design features/functions verified/validated by the specified test set;

3. path coverage: percentage of the total potential logic paths exercised by the specified test set;

4. node coverage: percentage of decision branch points exercised by the specified test set;

5. statement coverage: percentage of the total executable lines of code exercised by the specified test set; and

6. equivalence class coverage: percentage of equivalence classes exercised by the specified test set.

Almost any software unit of interest can become a target for test coverage measures. Coverage analysis provides a measure of completeness and also a measure of what software functionality can not be tested through dynamic test methods (C.3.4) and must be verified by other analysis methods.

c. Benefits: Coverage analysis provides a measure of test completeness and confidence that the software product will perform as expected in its operational as well as potentially abnormal operating environment. Such analysis also provides valuable understanding for how to optimizing test strategies to obtain the best coverage with limited personnel resources and schedule time.

d. Case Evidence: Coverage metrics such as requirements, feature, path, node, statement, equivalence class; test completeness metrics.

e. Limitations: Measuring coverage and analyzing any software that has not been adequately tested is effective when applied as part of most dynamic test methods (C.3.4). Limitations apply primarily to achieving high percentages of coverage due to limitations in test time and/or theoretical limits imposed by the mathematical nature of the problem being solved by the software implementation.

f. References:

1. Beizer, B., Software System Testing and Quality Assurance, International Thomson Computer Press, 1996. Defence Standard 00-42 (PART 2)/Issue 1, "Reliability And Maintainability Assurance Guides, Part 2: Software," United Kingdom Ministry of Defence, September 1997.

2. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

3. Kaner, C., Testing Computer Software, 2nd edition, John Wiley & Sons, 1993.

4. Kung, D., Hsia, P., and Gao, J., Testing Object-Oriented Software, IEEE Computer Society Press, 1998.

5. Perry, W., Effective Methods for Software Testing, 2nd edition, John Wiley & Sons, 1999.

6. RCTA/DO-178B/ED-12B, "Software Considerations in Airborne Systems and Equipment," Federal Aviation Administration software standard, RTCA Inc., December 1992.

C.3.4 Dynamic Test Methods

a. Purpose: Verify a system will meet stated software requirements through execution of the software program; detect any inherent defects in the software program.

b. Description: There are many dynamic testing methods. All these methods have varying specific objectives, but generally are intended to determine whether the software will fail during execution. Some methods focus on showing that the software will not fail, while other methods focus on trying to determine if the software will fail. Selecting which dynamic test methods are most effective for the specific application and provide the best case evidence that the software will not fail in its operational environment is part of a software engineering and reliability engineering task. Some of the specific dynamic test methods include: functional testing, interface testing, performance testing, probabilistic testing, regression testing, stress testing, and usability testing. Dynamic testing frameworks typically include a combination of these methods. Integrated with reliability failure analysis and modeling (C.1.14, C.1.20, C.3.9) as well as coverage analysis (C.3.3), a more effective testing strategy to achieve failure rate goals can be determined.

Functional testing verifies that the system delivers the functionality expected by the end user. Functional testing verifies that a system performs the functions described in the requirements specification. Functional test cases are generally developed from the requirements specification. Attention is paid to verifying that correct inputs produce correct outputs and that all functionality is present and fully implemented. Functional testing does not verify performance. Safety and security functionality is tested as part of the software functional requirements.

Interface testing verifies hardware/software, system software/application software, and application software/application software interfaces. Interface testing identifies potential failures resulting from interface design, data, and/or timing errors. Different types and ranges of parameters are passed under varying system loads and states. Snapshots of pre- and post-conditions are examined. Potential system integration and operational errors are detected prior to a system being fielded. Interface testing must be conducted in the operational environment, or a simulated operational environment, to yield valid results. Design by contract (C.2.1) provides a design basis for defining interface testing.

Performance testing exercises a system under varied loads, states, and modes to determine if response time, capacity, and throughput requirements will be met. Performance shortfalls are identified prior to a system being fielded. Performance testing must be conducted in the operational environment, or a simulated operational environment, for it to yield valid results.

Probabilistic testing provides input to a quantitative assessment of operational software reliability. During probabilistic or statistical-based testing, test cases are developed from operational profiles which reflect how different classes of users will use a system, the type and frequency of transactions performed, the anticipated system loading, and so forth. Greater weight is given to the correct operation of transactions that are performed frequently and considered essential than those that are performed infrequently and are not essential. This approach contrasts with typical software reliability models (C.1.14 and C.3.9) although probabilistic methods (C.1.9) along with probabilistic methods can be used to evaluate the validity of such models. Probabilistic testing yields reliability measures that correspond to how a system is expected to be used. Effective probabilistic testing is dependent on an accurate and complete set of operational profiles and the accuracy of parameter/data distribution assumptions.

Regression testing verifies changes (corrections, enhancements, adaptations) to the software have not introduced new errors and/or affected system performance, safety, reliability, or security. After a change is implemented, a subset of the original test cases is executed. The results are compared with the original results to ensure stable and predictable system behavior. Regression testing should be performed in conjunction with change impact analysis (C.1.1). Regression testing minimizes the potential for unexpected system behavior resulting from changes and/or enhancements. Test cases must be selected carefully and perhaps additional test cases constructed so that both local and global effects of the changes or enhancements are verified.

Stress testing determines: maximum peak loading conditions under which system will continue to perform as specified; and system overload conditions. It is essential to conduct stress testing to determine how a system will perform in the operational environment in relation to specified performance requirements. Typical stress testing emphasizes behavior at boundary and "out of boundary" conditions, so boundary value analysis (C.3.1) can provide guidance to constructing stress test cases. During stress testing system performance is monitored under low, normal, peak and overload conditions. Application functionality, interfaces, memory capacity, throughput capacity, and communication links are each "stressed" as appropriate to the application. Transaction rates are measured as the number of simultaneous users is increased. Stress testing provides a realistic assessment of how a system will perform in the operational environment under the most challenging conditions. It also helps identify conditions which may cause a system to enter an unknown or unsafe state.

Usability testing determines if a system performs in the operational environment in a manner acceptable to and understandable by the end-users. Usability testing, or Human Computer Interface (HCI) testing, is conducted by a team of end-users. The focus of usability testing is to verify that domain knowledge has been captured and implemented correctly; in particular in regard to: how a system will be used; what a system will be used for; and how end-users expect to interact with a system. The potential for induced or invited errors is examined in the context of end-user expectations. The potential for induced or invited errors and mismatches with end-user expectations/training are identified before a system is deployed. Many organizations attempt to execute the User's Manual and Operational Procedures as part of usability testing.

c. Benefits: Dynamic testing methods provide verification and validation confidence that requirements are met and that defects/faults have been eliminated prior to operational use.

d. Case Evidence: Coverage measures, fault/failure data for use in reliability estimation/prediction models, inputs to defect removal efficiency computations, demonstrated evidence that customer requirements have been met.

e. Limitations: Complete test coverage of all possible test cases is highly unlikely if not mathematically impossible. Testing is time consuming and resource intensive, so defect prevention and removal early in a software product's development or support cycle is more cost effective than "testing out" the defects.

f. References:

1. Beizer, B., Software System Testing and Quality Assurance, International Thomson Computer Press, 1996.

2. BS5760 Part 8: "Guide to the Assessment of Reliability of Systems Containing Software," British Standards Institution (BSI), October 1998.

3. Defence Standard 00-42 (PART 2)/Issue 1, “Reliability And Maintainability Assurance Guides, Part 2: Software,” United Kingdom Ministry of Defence, September 1997.

4. IEC 61508-7:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

5. Joint Software System Safety Committee and EIA G-46 Committee, "Software System Safety Handbook," Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.

6. Kaner, C., Testing Computer Software, 2nd edition, John Wiley & Sons, 1993.

7. Kung, D., Hsia, P., and Gao, J., Testing Object-Oriented Software, IEEE Computer Society Press, 1998.

8. Perry, W., Effective Methods for Software Testing, 2nd edition, John Wiley & Sons, 1999.

9. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw Hill, 1997 (4th Edition).

10. RCTA/DO-178B/ED-12B, “Software Considerations in Airborne Systems and Equipment,” Federal Aviation Administration software standard, RTCA Inc., December 1992.

11. System Safety Society, System Safety Analysis Handbook, July 1993.

C.3.5 Formal In-Process Reviews (Fagan Software Inspections)

a. Purpose: Identify errors in software engineering artifacts throughout the development life cycle

b. Description: Formal In-Process Reviews, or software inspections, are a rigorous examination of all software engineering artifacts which are undertaken to prevent, detect, and remove defects as early as possible in the life cycle. The formal process includes: defined participant roles; small inspection teams; preparation, inspection meeting, and follow-up procedures; metrics for person time spent, defects found, and variations against company and industry standards are collected. An inspection is conducted on specified software artifacts during each phase of the development life cycle and the results are compared to the previous phase for consistency, correctness, and completeness. Inspections identify defects in artifacts, deviations in consistency between artifacts, and the source of the deviation. In combination with Root Cause Analysis (C.3.10) areas of process improvement can be identified.

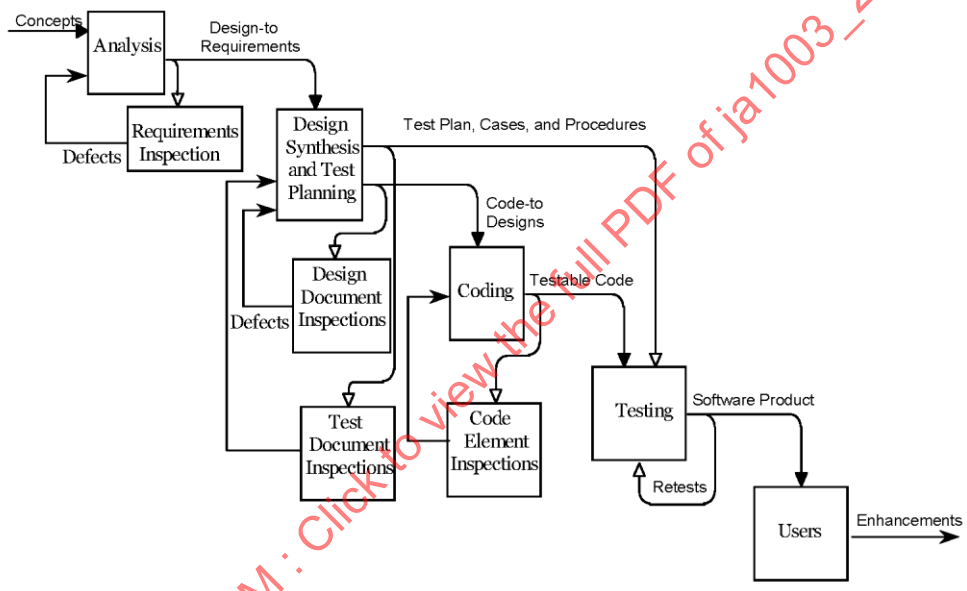

FIGURE C9—FORMAL IN-PROCESS REVIEWS ACROSS LIFE CYCLE ACTIVITIES

c. Benefits: Defects are identified and removed early in the life cycle, when it is schedule and cost effective to fix them. Recognized as a best practice by industry when conducted in accordance with the defined process.

d. Case Evidence: Measures of defects (number, severity, category) found and effort (preparation, meeting, follow-up) expended for a wide range of artifacts, including requirements, design, source code, test plans, build documentation, user guide, and system interface specifications. Supports the defect removal efficiency (C.1.3) measures.

e. Limitations: Formal In-Process Reviews (software inspections) are somewhat labor intensive, particularly inspections of large size source code implementations. Tailoring and selection of inspection artifacts is recommended for large projects. Inspections of software requirements, design, and test artifacts are most effective. Stratified inspections of software source code by criticality and feature coverage is also effective.

f. References:

1. Fagan, M. E., "Advances In Software Inspections," IEEE Transactions on Software Engineering, Vol. 12, No. 7, July 1986, pp 744-751.

2. Gilb, T. and Graham, D., Software Inspection, Addison-Wesley, 1993.

3. Ebenau, R. G., and Strauss, S. H., Software Inspection Process, McGraw-Hill, 1994.

4. IEC 61508:1998-09-15, "Functional safety of electrical/ electronic/ programmable electronic safety-related systems. Part 7: Overview of techniques and measures," International Electrotechnical Commission, 1998.

C.3.6 Operational Profile

a. Purpose: Define a quantitative characterization of how the software will be used.

b. Description: The reliability of a software-based product depends on how the computer and other external elements will use it. Making a good reliability estimate depends on testing the product as if it were in the field. The *operational profile*, a quantitative characterization of how the software will be used, is therefore essential to understand any software application. It is a fundamental concept which must be understood in order to apply Software Reliability Engineering (C.1.20) effectively and with any degree of validity. The operational profile has similar objectives as formal scenario analysis (C.1.5).

A *profile* is a set of independent possibilities called *elements*, and their associated probability of occurrence. If operation *A* occurs 60 % of the time, *B* occurs 30 %, and *C* occurs 10 %, for example, the profile is [*A*, 0.6...*B*, 0.3...*C*, 0.1]. The *operational* profile is the set of independent operations that a software system performs and their associated probabilities. Developing an operational profile for a system involves one or more of the five profiles illustrated in the figure below.

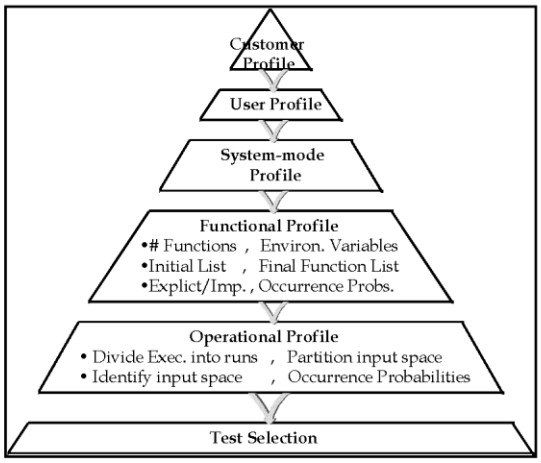

FIGURE C10—OPERATIONAL PROFILE DERIVATION

c. Benefits: The benefit-to-cost ration in developing and applying the operational profile is typically 10 or greater. This is accomplished through well-defined elicitation and documentation of customer operational use requirements, more efficient test selection that reduces test time, and more complete customer-based test coverage that reduces fielded faults. Cost to develop operational profiles is estimated at 1 % of software project cost or less with a savings in test cost alone at 10 % or more.

d. Limitations: It may be difficult to obtain the necessary information about system use from the customers. This is particularly the case for products that have multiple customers that span multiple domains of application. This does not mean operational profiles can not be developed, only that the possibility of missing situations of customer use can occur.

e. References:

1. Musa, John D. "Operational Profiles in Software Reliability Engineering," IEEE Software, March 1993, pages 14-32.

2. Musa, John D., Software Reliability Engineering, McGraw-Hill Book Company, NY, 1999.

3. Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

4. Lyu, Michael, Handbook of Software Reliability Engineering, McGraw Hill, 1996.

C.3.7 Peer Reviews

a. Purpose: Find and correct defects as early as possible in the development life cycle through review by peer experts that may be internal or external to the organization.

b. Description: Any type of technical or management product or process may be reviewed, whether it is a deliverable to the customer or internal to the project and whether it is in an intermediate or final state. These reviews and inspections are interactive, with a focus on constructive criticism. Peer reviews focus on the work product being reviewed and not on the author of that product. Peer reviews are normally conducted to certify products prior to their proceeding to the next phase of development. These structured examinations are used to verify correctness and completeness of products or processes. They are useful when feedback is required from cross-functional or cross-organizational groups, the scope of the feedback is relatively large, the product or process is relatively complex or has a far-reaching impact, or when education of a group is necessary. The common features of the peer review process include: (1) a triggering event, (2) identification of the review participants and their role in the review as either facilitator, author, reviewer, or recorder, (3) a schedule for the review, (4) distribution of the product to be reviewed prior to the peer review meeting, and (5) the actual review, with action items assigned for any unresolved issues or questions identified during the review. All action items have a designated assignee and required completion/re-review date. It is common for defects identified during a peer review to be classified according to type, source, and severity to foster continuous process improvement. Formal in-process review (C.3.5) is one particular form of peer review. The certification authority for a product may require several formal peer reviews by external experts prior to product acceptance.

c. Benefits: More and different types of defects are identified because peer reviews involve multiple stakeholders from different perspectives. Defects are identified early in the life cycle when it is more cost effective to fix them.

d. Case Evidence: Review reports provide directed evidence for use in verifying and/or validating that reliability requirements and/or activities are being adequately accomplished.

e. Limitations: Peer reviews are generally not useful for identifying performance defects, but may identify problems that could affect performance.

f. References:

1. Ebenau, R. and Strauss, S., Software Inspection Process, McGraw-Hill, 1994.

2. Gilb, T. and Graham, D., Software Inspection, Addison-Wesley, 1993.

3. Kan, S.H., Metrics and Models in Software Quality Engineering, Addison-Wesley Publishing Co., 1995.

4. IEEE Std-1028-1994,"IEEE Standard for Software Reviews," IEEE Computer Society, December 1997.

C.3.8 Reliability Bench Marking

a. Purpose: Process used in identifying gap(s) and improvement opportunities for target products compared to "Best-In-Class" products with respect to reliability metrics.

b. Description: Reliability benchmarking activities may include benchmarking metric selection, test design and planning, failure criteria definition, sample size determination, data collection and analysis, assessment of design differences that could affect product reliability, and use of this information in a Benchmarking process. In general, Benchmarking is the search for best practices that, when applied, lead to superior performance. The application of Benchmarking study findings should produce increased customer satisfaction, improve competitive advantage, and shorten product development cycle time.

   Benchmarking should be performed early in the product development phase so that findings can be implemented and reliability can be designed into the product. Reliability Benchmarking can be applied in circumstances such as the following:

   1. when there is an unexpected decreasing trend of products' reliability performance and customer satisfaction;

   2. when an organization wants to improve the reliability performance of their products so that they become competitive with the best products in the industry;

   3. when it is important to consider alternative designs for a specific product or search for best solutions relating to design concerns; and

   4. when it is important to determine the upper limits of reliability performance that can be expected by comparison with the "Best-In-Class" products in a similar application domain;

c. Benefits: Provides data and quantitative evidence to convince management of a gap that must be closed; identifies design leverage, new technology, and quality issues for value-added engineering and continuous improvement; provides reliability-related information for system/software design specification, key testing concepts, and reliability target setting.

d. Case Evidence: Reliability metrics and methods considered to be "Best-In-Class"

e. Limitations: May be difficult to get benchmarking data for systems/software in a similar application domain; may lead to a "follower" mindset rather than attempting to do what is appropriate for the specific application.

f. References:

1. Basili, Vic, Boehm, Barry, and others, "What We Have Learned About Fighting Defects," Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS™02), IEEE Computer Society, 2002. http://www.CeBASE.org

2. Defence Standard 00-42 (PART 2)/Issue 1, “Reliability And Maintainability Assurance Guides, Part 2: Software,” United Kingdom Ministry of Defence, September 1997.

3. Dujmovid, J., "Quantitative Methods for Design of Benchmark Suites," Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '96), IEEE, pp 162-166, 1996.

4. Mukherjee,A. and Siewiorek, D.P., "Measuring Software Dependability by Robustness Benchmarking," IEEE Transactions on Software Engineering, Vol. 23, No. 6, June 1997.

C.3.9 Reliability Estimation Modeling

a. Purpose: Estimate software reliability for either the present or at some future time based on historical, test, and/or operational failure data.

b. Description: Estimation models apply statistical techniques to the observed failures during software testing and operation to forecast the product's reliability. Musa's Software Reliability Engineering method (C.1.20, Appendix D) provides a comprehensive approach to the application of reliability estimation modeling. A generic 11-step process can be followed for estimating software reliability. These steps should be tailored to the application project and the current life cycle phase:

- identify the software application being evaluated;
- specify the reliability requirement for this software;
- allocate the reliability requirement;
- define failure categories and conditions;
- define operational environment and profile(s);
- define test cases and procedures that correspond to the operational environment and profile(s);
- select appropriate software reliability models;
- collect data from test results;
- estimate parameters from historical data;
- validate the model; and

- perform the analysis.

Some of the more common reliability estimation models include: Duane, General Exponential, Musa basic, Musa logarithmic, Littlewood/Verrall, and Schneidewind.

TABLE C5—COMMON SOFTWARE RELIABILITY ESTIMATION MODELS

Model Name	Formula for Failure Rate	Estimation/data required	Limitations and Constraints
Musa Basic	$\lambda(t) = \lambda_0 \exp[-(\lambda_0/\nu_0)t]$ or $\lambda(\mu) = \lambda_0[1-\mu/\nu_0]$	Estimate of initial failure rate λ_0 Estimate of total expected failures ν_0 Number of detected faults μ at some time t	Software must be exercised Assumes no new faults introduced in correction Assumes number of residual faults decreases linearly over time
Musa Logarithmic	$\lambda(t) = \lambda_0/(\lambda_0\theta t + 1)$ or $\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$	Estimate of initial failure rate λ_0 Estimate of failure intensity decay θ Number of detected faults μ at time t	Software must be exercised Assumes no new faults introduced in correction Assumes number of residual faults decreases exponentially over time
Schneidewind	$\alpha \exp(-\beta i)$	Faults detected in equal interval i Estimation of failure rate at start of first interval Estimation of proportionality constant of failure rate over time β	Software must be exercised Assumes no new faults introduced in correction Assumes number of residual faults decreases exponentially over time
Littlewood/ Verrall	$\alpha/[t + \Psi(l)]$	Estimate of number of failures α Estimate of reliability growth Ψ	Software must be exercised Assumes uncertainty in correction process

c. Benefits: These models are useful in estimating the current software failure rate and whether this failure rate is within the software reliability requirement and associated confidence limits. Estimates of the future software failure rate/reliability can be made based on continued defect removal through testing or maintenance support. Model accuracy can be estimated based on statistical analysis (C.1.21) and probabilistic methods (C.1.9). Estimates can be made for how many corrective maintenance changes and the level of associated support resources that can be expected if the product is fielded at the existing level of software reliability.

d. Case Evidence: Current software reliability, accuracy/confidence in software reliability estimates, estimate of additional testing to reach software reliability requirement and/or goal, verification of conformance to acceptance criteria, safety/security certification evidence.

e. Limitations: All of these models are used late in the life cycle so correcting requirements or design deficiencies to improve reliability is not a cost-effective option. These models do not identify how to improve reliability. Model assumptions may not hold true such as the assumption that no new faults are introduced during fault correction.

f. References:

1. ANSI/AIAA R-013-1992, “AIAA Recommended Practice for Software Reliability,” February 1993.

2. BS5760 Part 8: "Guide to the Assessment of Reliability of Systems Containing Software," British Standards Institution (BSI), October 1998.

3. Defence Standard 00-42 (PART 2)/Issue 1, “Reliability And Maintainability Assurance Guides, Part 2: Software,” United Kingdom Ministry of Defence, September 1997.

4. IEEE Std. 982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE, 1988.

5. IEEE Std. 982.2-1988, "IEEE Guide for the Use of the Standard Dictionary of Measures to Produce Reliable Software," IEEE, 1988.

6. Lyu, M., Handbook of Software Reliability Engineering, McGraw-Hill, 1996.

7. Musa, J., Software Reliability Engineering, McGraw-Hill, 1999.

8. Peters, J., Software Engineering: An Engineering Approach, John Wiley & Sons, 1999.

C.3.10 Root Cause Analysis

a. Purpose: Determine the initiating event that caused a system to fail and/or a defect to be introduced in the system implementation.

b. Description: Root cause analysis is an investigative technique that is used to determine how, when, and why a defect was introduced and why it escaped detection in earlier phases. Root cause analysis is conducted by examining a defect, then tracing back step by step through the design and the decisions and assumptions that supported the design to the source of the defect. Root cause analysis can be applied to any situation, product, or process problem where the intent is to not only fix a specific instance of the problem, but to fix the all potential instances of the problem within the application population through development of a comprehensive corrective action. Root cause analysis supports defect prevention and continuous process improvement.

Root cause analysis typically consists of four steps:

1. gather data about the problem;

2. analyze the data for possible root causes, solutions, and other potential instances of the problem;

3. communicate prevention action plans for decision; and

4. implement the recommended action.

As data is gathered about a particular problem, it is important to determine the origin and scope of the problem and determine if there are any other potential parts of the software where this problem might also exist. Since problems may well involve interactions with system components other than the target software, analyzing the data should involve a cross-functional team to determine what data to collect and what analysis technique(s) to use. The cross-functional team will provide effective solutions where the root cause involves multiple components as well as the software. Once the root cause is identified and verified, it is important to consider whether there are organizational process improvements that might prevent future recurrence of the same problem in the same or other projects. A proper root cause analysis process should identify where in the organization a policy/procedure can be established, or where an existing policy/procedure can be better implemented to keep the problem from recurring. Quality Function Deployment (C.1.11) is a technique that can be used to implement a root cause analysis process and represent the results as quality improvements.

c. Benefits: Root cause analysis is a disciplined technique for solving problems using a pre-defined, proven process. Historical data provides evidence of process improvement. The process of conducting root cause analysis may uncover defects in other areas as well. Supports multiple defect removal and future prevention of similar defects through process improvement. This is particularly important within critical systems involving safety, security, and/or high reliability assurance.

d. Case Evidence: Evidence of specific defect removal and root cause source identification; supports defect removal efficiency computation; provides qualitative confidence that all common cause defects have been removed.

e. Limitations: Root cause analysis can be time consuming on large complex systems, particularly when resources are limited.

f. References:

1. Akao, Y., Quality Function Deployment, Productivity Press, January 1990.

2. Akao, Y. and Mizuno, S., QFD: The Customer-Driven Approach to Quality Planning and Development, Productivity Press, April 1994.

3. Root Cause Analysis: Improving Performance for Bottom Line Results, Reliability Center, Inc., 501 Westover Avenue, Hopewill, VA 23860, 1999.

4. Root Cause Analysis Handbook, ABS Group, Inc., 1000 Technology Drive, Knoxville, TN 37932, 1999.

C.3.11 Testability Analysis, Fault Injection, Failure Assertion

a. Purpose: Determine if a system design can be verified and is supportable.

b. Description: Testability analysis began as a research and development project in the late 1970s. The goal was to derive an indicator of the testability of a software product from an analysis of the controllability and observability of internal nodes. This indicator was based on measurements of the number of unique operators, number of unique operands, total occurrence of each operator, total occurrence of each operand, and number of unique logic paths. This algorithmic analysis uncovers potential faults in the form of unreachable nodes, unused nodes, and non-deterministic conditions. Since the original project, testability analysis has been expanded to include analyses of traceability, repeatability, predictability, functional testability, accessibility, fault injection, and failure assertion.

Another aspect of testability analysis is the study of requirements specifications to determine if they are testable–that is, whether tests or other analysis methods can produce convincing evidence that the requirements have been met. If it is determined that the requirements are either ill-specified or there are no practical methods to verify or validate that the requirements can be met, then the requirements statements in question may be appropriately modified to be testable.

c. Benefits: Testability analysis will help identify whether or not requirements or a system design can be verified, and if not, the requirements and/or design modules that need to be modified. Because this occurs prior to the testing phase the cost of rework is reduced. This cost savings extends to the operations and support phase as well.

d. Case Evidence: Removal of potential faults and improvement of the defect removal efficiency measure.

e. Limitations: Testability analysis is most useful when applied to large complex systems.

f. References:

1. Beizer, B., Software System Testing and Quality Assurance, International Thomson Computer Press, 1996.

2. Friedman, M. and Voas, J., Software Assessment: Reliability, Safety, and Testability, John Wiley & Sons, 1995.

3. Kan, S.H., Metrics and Models in Software Quality Engineering, Addison-Wesley Publishing Co., 1995.

4. Kaner, C., Testing Computer Software, 2nd edition, John Wiley & Sons, 1993.

5. Kung, D., Hsia, P., and Gao, J., Testing Object-Oriented Software, IEEE Computer Society Press, 1998.

6. Parkinson, J.S., Classification of Programmable Electronic Systems Operation for Testability, Directions in Safety-Critical Systems, Springer-Verlag, 1993, p. 67-83.

7. Perry, W., Effective Methods for Software Testing, 2nd edition, John Wiley & Sons, 1999.

8. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw Hill, 1997 (4th Edition).

C.4 Management Techniques

C.4.1 Configuration Management

a. Purpose: Identify system items requiring control (configuration items), control the release and change of these items, record and report the status of the items and change requests, and verify the completeness and correctness of the items.

b. Description: Configuration Management (CM) is the process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system lifecycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items. Configuration Management is practiced in one form or another as part of any software engineering project where several individuals or organizations have to coordinate their activities. While the basic disciplines of Configuration Management are common to both hardware and software engineering projects, there are some differences in emphasis due to the nature of software products. [3]

Software Configuration Management (SCM) is a system for managing the evolution of software products, both during the initial stages of development and during all stages of maintenance. A software product encompasses the complete set of computer programs, procedures, and associated documentation and data designated for delivery to a user. All supporting software used in development, even though not part of the software product, should also be controlled by SCM.

The SCM system is the collection of activities performed during a software engineering project to:

- determine and identify those entities of the software product that need to be controlled;
- ensure those entities have necessary and accurate definitions and documentation;
- ensure changes are made to the entities in a controlled manner;
- ensure that the correct version of the entities/software product are being used; and
- ascertain, at any point in time, the status of an entity (e.g., whether a specific entity is completed, being changed, waiting to be tested, or released to the customer).

SCM is performed within the context of several basic configuration management functions, including:

- Configuration Identification;
- Configuration Control (change control)
- Configuration Status Accounting
- Audits and Reviews
- Release Processing

SCM provides a common point of integration for all planning, oversight, and implementation activities for a software project or product line. It provides the framework (labeling and identification) for interfacing different activities and defining the mechanisms (change controls) necessary for coordinating parallel activities of different groups. SCM also provides a framework for controlling computer program interfaces with their underlying support hardware, and coordinating software changes when both hardware and software may be evolving during development or maintenance activities. SCM provides management with the visibility (through status accounting and audits) of the evolving software products that make technical and managerial activities more effective.

c. Benefits: Provides baseline identification of development and released software product; provides a snapshot of dynamically changing software; tracks concurrent modification of modules or components; ensures the orderly release and implementation of new or revised software products.

d. Limitations: For larger projects, configuration management requires automated tools to adequately implement version control and issue tracking along with a link to the operational change request/failure reporting system.

e. Case Evidence: Software product version identification, software artifact version identification, issue tracking metrics throughout life cycle, key process link between change management and FRACAS (C.4.2) system.

f. References:

1. ANSI/IEEE Std 1042-1993, "IEEE Guide to Software Configuration Management, " IEEE Computer Society, 1993.

2. CMMI-SE/SW-Continuous, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Continuous Representation," CMU/SEI-2000-TR-019, November 2000.

3. CMMI-SE/SW-Staged, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Staged Representation," CMU/SEI-2000-TR-018, November 2000.

4. Hass, Anne M J, Configuration Management Principles and Practice, Addison Wesley, 2003.

5. SQAS20.01.00-2000, "Software Configuration Management (SCM): A Practical Guide," United States Department of Energy, Quality Managers Software Quality Assurance Subcommittee, April, 2000. http://cio.doe.gov/sqas/

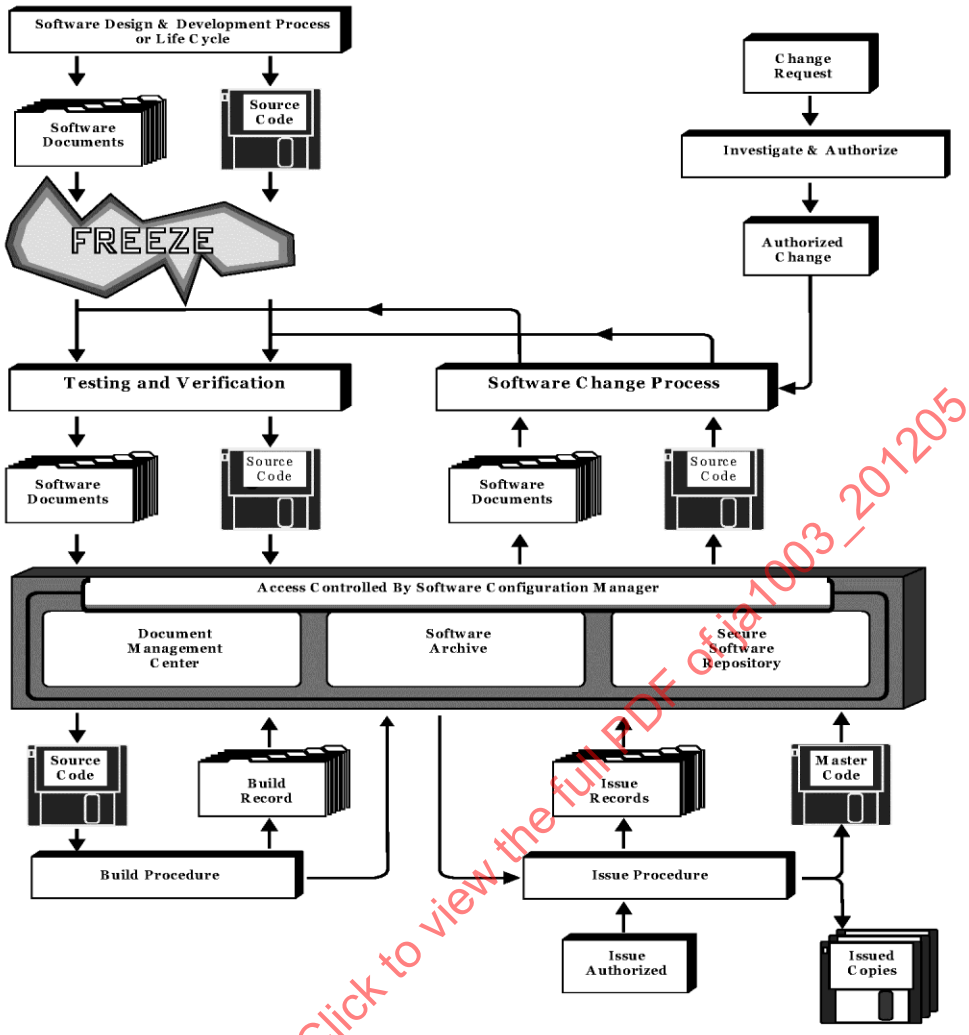

FIGURE C11—ELEMENTS OF A SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

C.4.2 Failure Reporting Analysis And Corrective Action System (FRACAS)

a. Purpose: The purpose of a FRACAS is to document, track, analyze and correct reported failures in a closed loop fashion. It is also a management tool for focusing on critical concerns, resolution and resources dedication for timely program execution.

b. Description: A FRACAS is normally implemented at the system, hardware and software level and typically involves more than one physical entity. Ideally, however, the system level FRACAS will coordinate the failures that are recorded between the hardware and software. The software FRACAS is similar to a hardware FRACAS in many ways. Failures are recorded, corrected and root causes of the failures analyzed. This is typically done by a Failure Review Board. While hardware failures may be analyzed to find product specific defects, software failures are analyzed more to find deficiencies in the software engineering process. A hardware FRACAS will track information that is pertinent to correcting the hardware failure. A software FRACAS will track information that is pertinent to reproducing the software failure so that it can be corrected. It will also track the root cause of each failure so that a Pareto analysis (C.1.8) can be performed on a group of failures at some later time.

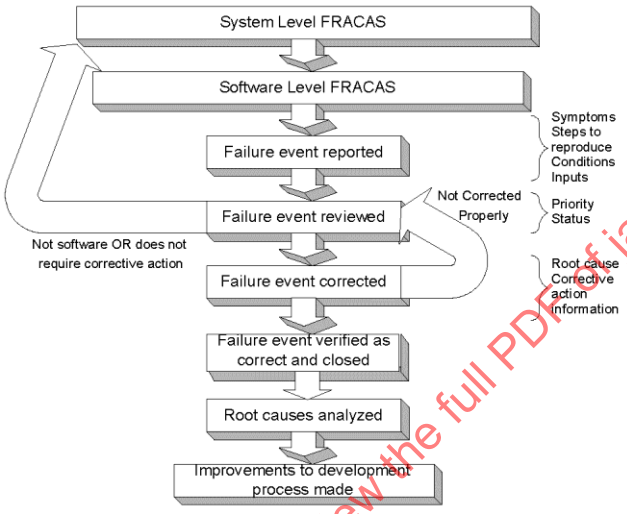

FIGURE C12—ELEMENTS OF A SYSTEM-SOFTWARE FRACAS

c. Benefits: A FRACAS: provides a single reliability data source that permits quick and efficient access to reliability data; supports for root cause investigation to resolve concerns; documents the initial incident to the verification of the corrective action; and builds on past experience and provides better future reliability targets and risk assessment.

d. Case Evidence: Failure data (testing and operational field), root causes of failures.

e. Limitations: A FRACAS: does not prevent failure from occurring in the first place; requires resources that could be used for failure prevention; may yield false quality perspective if the data is not properly selected or is exposed to noise.

f. References:

1. Lakey, Peter and Neufelder, Ann Marie, "System and Software Reliability Assurance Notebook," Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997. http://www.cs.colostate.edu/~cs530/rh/

2. MIL-HDBK-2155, "Failure Reporting, Analysis and Corrective Action Taken," Department of Defense.

C.4.3 Life Cycle Process Standard

a. Purpose: Provide a set of activities that enable the repeatable definition, development, and support of a software product within the context of its system application.

b. Description: A life cycle process standard defines what activities are to be conducted throughout the definition, development, and support of a software product. The activities include typical process inputs, procedural steps, and outputs along with the policies that constrain the activities and resources needed to accomplish the activities. The organization of the activities into sequential phases conducted within a defined time period/schedule and with planned resources defines the specific project life cycle model that is to be used. Typical core software engineering activities include: requirements analysis, design, implementation, test, and change modification. There are many software management activities that support the core activities, such as project management, risk management, logistics management, configuration management, quality engineering/assurance, and specialty engineering (e.g., engineering support in the areas of reliability, maintainability, safety, security, logistics). Systems engineering and associated life cycle process standard(s) should include an integration of applicable hardware and software life cycle process standards. Other enterprise management activities may include marketing, business planning, and other such intergroup corporate functions.

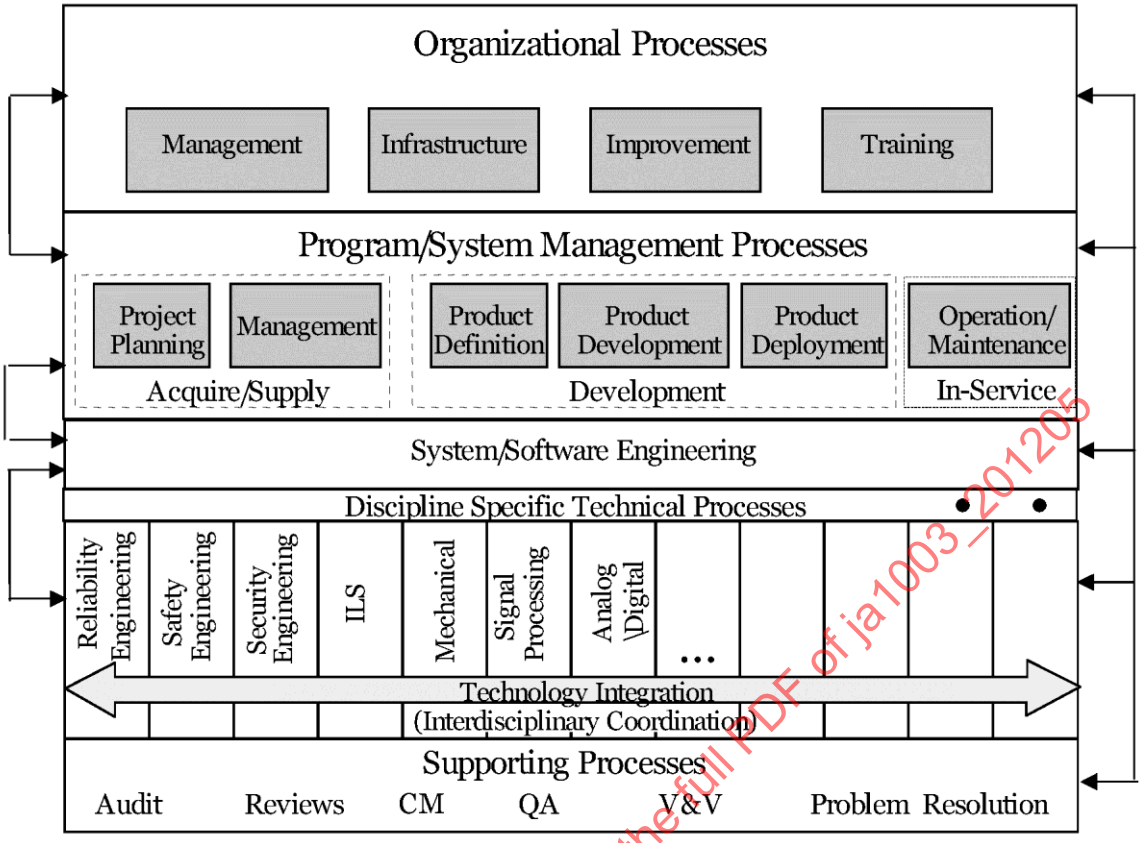

FIGURE C13—EXAMPLE INTEGRATED LIFE CYCLE MANAGEMENT SYSTEM

c. Benefits: A well-defined life cycle process standard, when appropriately tailored for application use, provides confidence that the customer's requirements have been defined, activities are conducted that provide evolutionary evidence that the requirements are being met, and evidence exists that demonstrates the requirements have been met for the initial delivery to the customer as well as any subsequent updated releases. A well-defined life cycle process standard establishes the core engineering base upon which a software reliability program can be integrated. Without this core engineering base it is unlikely that a software reliability program will be effective.

d. Case Evidence: Life cycle activities that are core elements of a software reliability program can be defined as part of the software reliability plan and assessment evidence provided in the case as to whether those activities have been conducted in accordance with the life cycle process standard.

e. Limitations: Cost, schedule and performance balance may require tailoring of the selected life cycle process standard; tradeoffs often affect reliability.

f. References:

1. CMMI-SE/SW-Continuous, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Continuous Representation," CMU/SEI-2000-TR-019, November 2000.

2. CMMI-SE/SW-Staged, V1.02, "CMMI for Systems Engineering/Software Engineering, Version 1.02, Staged Representation," CMU/SEI-2000-TR-018, November 2000.

3. ANSI/IEEE Std 830-1998, "IEEE Recommended Practice for Software Requirements Specifications, " IEEE Computer Society, 1998.

4. IEEE Std-1220-1998, "IEEE Standard for Application and Management of the Systems Engineering Process," IEEE Computer Society, December 1998.

5. IEEE/EIA Std 12207.0-1996, “Software life cycle processes,” IEEE Computer Society, March 1998.

6. ISO/IEC 12207, “Software Life Cycle Processes,” August 1, 1995.

7. ISO/IEC 15288, “Systems Engineering – System Life Cycle Processes,” Edition 1, November 8, 2002.

8. ISO/TR 15497, "Development Guidelines for Vehicle Based Software, the Motor Industry," Motor Industry Software Reliability Association, ISBN 0 9524156 0 7, November 1994.

9. Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw Hill, 1997 (4th Edition).

10. RCTA/DO-178B/ED-12B, “Software Considerations in Airborne Systems and Equipment,” Federal Aviation Administration software standard, RTCA Inc., December 1992.

C.4.4 Process Assessment

a. Purpose: Process assessment helps software organizations improve by identifying critical problems with their software processes and establishing improvement areas and priorities.

b. Description: A software process assessment is a review of a software organization to advise its management and professionals on how they can improve their operation. A software process audit is a review to determine whether there are gaps in how an organization follows its officially approved process. Both types of reviews are important to provide evidence of software engineering capability. Both types of reviews can be conducted using a wide range of evaluators. Evaluators may be independent, non-independent, external, internal, requested by management, imposed by an outside customer or certification authority, or some combination. One common characteristic of evaluators is that they should be software professionals, knowledgeable in the targeted application areas, and trained in the assessment/audit process to be applied. The process assessment is just part of a comprehensive Software Process Improvement (SPI) approach.

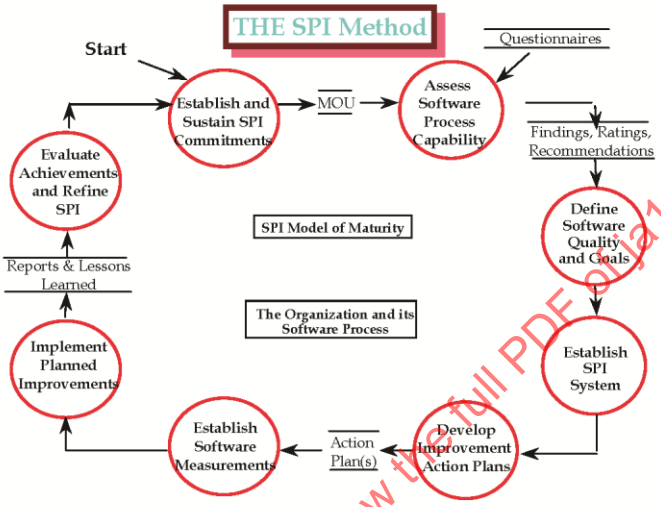

FIGURE C14—EXAMPLE SOFTWARE PROCESS IMPROVEMENT CYCLE