BS EN 62628:2012

BSI Standards Publication

Guidance on software aspects of dependability

bsi.

...making excellence a habit.™

National foreword

This British Standard is the UK implementation of EN 62628:2012. It supersedes BS EN 60730-8:1998 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee DS/1, Dependability.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Published by BSI Standards Limited 2012

ISBN 978 0 580 76381 6

ICS 03.120.01

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 October 2012.

Amendments issued since publication

Amd. No.	Date	Text affected

EUROPEAN STANDARD

NORME EUROPÉENNE

EUROPÄISCHE NORM

EN 62628

September 2012

ICS 03.120.01

English version

Guidance on software aspects of dependability
(IEC 62628:2012)

Lignes directrices concernant la sûreté de fonctionnement du logiciel
(CEI 62628:2012)

Leitlinien zu Softwareaspekten der Zuverlässigkeit
(IEC 62628:2012)

This European Standard was approved by CENELEC on 2012-09-12. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.

CENELEC

European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Management Centre: Avenue Marnix 17, B - 1000 Brussels

Ref. No. EN 62628:2012 E

Foreword

The text of document 56/1469/FDIS, future edition 1 of IEC 62628, prepared by IEC/TC 56, "Dependability" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN 62628:2012.

The following dates are fixed:

• latest date by which the document has to be implemented at national level by publication of an identical national standard or by endorsement	(dop)	2013-06-12
• latest date by which the national standards conflicting with the document have to be withdrawn	(dow)	2015-09-12

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC [and/or CEN] shall not be held responsible for identifying any or all such patent rights.

Endorsement notice

The text of the International Standard IEC 62628:2012 was approved by CENELEC as a European Standard without any modification.

In the official version, for Bibliography, the following notes have to be added for the standards indicated:

IEC 62508	NOTE	Harmonized as EN 62508.
IEC 60300-1	NOTE	Harmonized as EN 60300-1.
IEC 60300-2	NOTE	Harmonized as EN 60300-2.
IEC 60300-3-3	NOTE	Harmonized as EN 60300-3-3.
IEC 62347	NOTE	Harmonized as EN 62347.
IEC 61160	NOTE	Harmonized as EN 61160.
IEC 61078	NOTE	Harmonized as EN 61078.
IEC 61025	NOTE	Harmonized as EN 61025.
IEC 61165	NOTE	Harmonized as EN 61165.
IEC 62551[1)]	NOTE	Harmonized as EN 62551[1)].
IEC 60812	NOTE	Harmonized as EN 60812.
IEC 60300-3-1	NOTE	Harmonized as EN 60300-3-1.
IEC 61508-3	NOTE	Harmonized as EN 61508-3.
IEC 62429	NOTE	Harmonized as EN 62429.
IEC 61014	NOTE	Harmonized as EN 61014.
IEC 61164	NOTE	Harmonized as EN 61164.
IEC 62506[1)]	NOTE	Harmonized as EN 62506[1)].

[1)] To be published.

Annex ZA
(normative)
Normative references to international publications with their corresponding European publications

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

NOTE When an international publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

Publication	Year	Title	EN/HD	Year
IEC 60050-191	-	International Electrotechnical Vocabulary (IEV) - Chapter 191: Dependability and quality of service	-	-
IEC 60300-3-15	-	Dependability management - Part 3-15: Application guide - Engineering of system dependability	EN 60300-3-15	-

CONTENTS

INTRODUCTION

Software has widespread applications in today's products and systems. Examples include software applications in programmable control equipment, computer systems and communication networks. Over the years, many standards have been developed for software engineering, software process management, software quality and reliability assurance, but only a few standards have addressed the software issues from a dependability perspective.

Dependability is the ability of a system to perform as and when required to meet specific objectives under given conditions of use. The dependability of a system infers that the system is trustworthy and capable of performing the desired service upon demand to satisfy user needs. The increasing trends in software applications in the service industry have permeated in the rapid growth of Internet services and Web development. Standardized interfaces and protocols have enabled the use of third-party software functionality over the Internet to permit cross-platform, cross-provider, and cross-domain applications. Software has become a driving mechanism to realize complex system operations and enable the achievement of viable e-businesses for seamless integration and enterprise process management. Software design has assumed the primary function in data processing, safety monitoring, security protection and communication links in network services. This paradigm shift has put the global business communities in trust of a situation relying heavily on the software systems to sustain business operations. Software dependability plays a dominant role to influence the success in system performance and data integrity.

This International Standard provides current industry best practices and presents relevant methodology to facilitate the achievement of software dependability. It identifies the influence of management on software aspects of dependability and provides relevant technical processes to engineer software dependability into systems. The evolution of software technology and rapid adaptation of software applications in industry practices have created the need for practical software dependability standard for the global business environment. A structured approach is provided for guidance on the use of this standard.

The generic software dependability requirements and processes are presented in this standard. They form the basis for dependability applications for most software product development and software system implementation. Additional requirements are needed for mission critical, safety and security applications. Industry specific software qualification issues for reliability and quality conformance are not addressed in this standard.

This standard can also serve as guidance for dependability design of firmware. It does not however, address the implementation aspects of firmware with software contained or embedded in the hardware chips to realize their dedicated functions. Examples include application specific integrated circuit (ASIC) chips and microprocessor driven controller devices. These products are often designed and integrated as part of the physical hardware features to minimize their size and weight and facilitate real time applications such as those used in cell phones. Although the general dependability principles and practices described in this standard can be used to guide design and application of firmware, specific requirements are needed for their physical construction, device fabrication and embedded software product implementation. The physics of failure of application specific devices behaves differently as compared to software system failures.

This International Standard is not intended for conformity assessment or certification purposes.

GUIDANCE ON SOFTWARE ASPECTS OF DEPENDABILITY

1 Scope

This International Standard addresses the issues concerning software aspects of dependability and gives guidance on achievement of dependability in software performance influenced by management disciplines, design processes and application environments. It establishes a generic framework on software dependability requirements, provides a software dependability process for system life cycle applications, presents assurance criteria and methodology for software dependability design and implementation and provides practical approaches for performance evaluation and measurement of dependability characteristics in software systems.

This standard is applicable for guidance to software system developers and suppliers, system integrators, operators and maintainers and users of software systems who are concerned with practical approaches and application engineering to achieve dependability of software products and systems.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 60050-191, *International Electrotechnical Vocabulary – Chapter 191: Dependability and quality of service*

IEC 60300-3-15, *Dependability management – Part 3-15: Application guide – Engineering of system dependability*

3 Terms, definitions and abbreviations

For the purposes of this document, the terms and definitions given in IEC 60050-191, as well as the following apply.

3.1 Terms and definitions

3.1.1
software
programs, procedures, rules, documentation and data of an information processing system

Note 1 to entry: Software is an intellectual creation that is independent of the medium upon which it is recorded.

Note 2 to entry: Software requires hardware devices to execute programs and to store and transmit data.

Note 3 to entry: Types of software include firmware, system software and application software.

Note 4 to entry: Documentation includes: requirements specifications, design specifications, source code listings, comments in source code, “help” text and messages for display at the computer/human interface, installation instructions, operating instructions, user manuals and support guides used in software maintenance.

3.1.2
firmware
software contained in a read-only memory device, and not intended for modification

EXAMPLE Basic input/output system (BIOS) of a personal computer.

Note 1 to entry: Software modification requires the hardware device containing it to be replaced or re-programmed.

3.1.3
embedded software
software within a system whose primary purpose is not computational

EXAMPLES Software used in the engine management system or brake control systems of motor vehicles.

3.1.4
software unit
software module
software element that can be separately compiled in programming codes to perform a task or activity to achieve a desired outcome of a software function or functions

Note 1 to entry: The terms "module" and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

Note 2 to entry: In an ideal situation, a software unit can be designed and programmed to perform exactly a specific function. In some applications, it may require two or more software units combined to achieve the specified software function. In such cases, these software units are tested as a single software function.

3.1.5
software configuration item
software item that has been configured and treated as a single item in the configuration management process

Note 1 to entry: A software configuration item can consist of one or more software units to perform a software function.

3.1.6
software function
elementary operation performed by the software module or unit as specified or defined as per stated requirements

3.1.7
software system
defined set of software items that, when integrated, behave collectively to satisfy a requirement

EXAMPLES Application software (software for accounting and information management); programming software (software for performance analysis and CASE tools) and system software (software for control and management of computer hardware system such as operating systems).

3.1.8
software dependability
ability of the software item to perform as and when required when integrated in system operation

3.1.9
software fault
bug
state of a software item that may prevent it from performing as required

Note 1 to entry: Software faults are either specification faults, design faults, programming faults, compiler-inserted faults or faults introduced during software maintenance.

Note 2 to entry: A software fault is dormant until activated by a specific trigger, and usually reverts to being dormant when the trigger is removed.

Note 3 to entry: In the context of this standard, a bug is a special case of software fault also known as latent software fault.

3.1.10
software failure
failure that is a manifestation of a software fault

Note 1 to entry: A single software fault will continue to manifest itself as a failure until it is removed.

3.1.11
code
character or bit pattern that is assigned a particular meaning to express a computer program in a programming language

Note 1 to entry: Source codes are coded instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator.

Note 2 to entry: Coding is the process of transforming of logic and data from design specifications or descriptions into a programming language.

Note 3 to entry: A programming language is a language used to express computer programs.

3.1.12
(computer) program
set of coded instructions executed to perform specified logical and mathematical operations on data

Note 1 to entry: Programming is the general activity of software development in which the programmer or computer user states a specific set of instructions that the computer must perform.

Note 2 to entry: A program consists of a combination of coded instructions and data definitions that enable computer hardware to perform computational or control functions.

3.2 Abbreviations

ASIC	Application specific integrated circuit
CASE	Computer-aided software engineering
CMM	Capability maturity model
CMMI	Capability maturity model integration
COTS	Commercial-off-the-shelf
FMEA	Failure mode and effects analysis
FTA	Fault tree analysis
IP	Internet protocol
IT	Information technology
KSLOC	Kilo-(thousand) source lines of code
ODC	Orthogonal defect classification
RBD	Reliability block diagram
USB	Universal serial bus

4 Overview of software aspects of dependability

4.1 Software and software systems

Software is a virtual entity. In the context of this standard, software refers to procedures, programs, codes, data and instructions for system control and information processing. A software system consists of an integrated collection of software items such as computer programs, procedures, and executable codes, and incorporated into physical host of the processing and control hardware to realize system operation and deliver performance functions. The hierarchy of the software system can be viewed as a structure representing the system architecture and consisting of subsystem software programs and lower-level software units. A software unit can be tested as specified in the design of a program. In some cases,

two or more software units are required to construct a software function. The system encompasses both hardware and software elements interacting to provide useful functions in rendering the required performance services.

In a combined hardware/software system, the software elements of the system contribute in two major roles: a) operating software to run continuously to sustain hardware elements in system operation; and b) application software to run as and when required upon user demands for provision of specific customer services. Dependability analysis of the software sub-systems has to consider the software application time factors in the system operational profile and those software elements required for full-time system operation. Software modelling is needed for reliability allocation and dependability assessment of software-based systems.

Human aspects of dependability [1][1] play a pivotal role in guiding effective software design and implementation. The human-machine interface and operating environment influence the outcome of software and hardware interaction and affect the dependability of system performance. This leads to a strategic need for software dependability design and perfective maintenance efforts in the software life cycle process [2].

4.2 Software dependability and software organizations

Software dependability is achieved by proper design and appropriate incorporation into system operation. This standard presents an approach where existing dependability techniques and established industry best practices can be identified and used for software dependability design and implementation. The dependability management systems [3, 4] describe where relevant dependability activities can be effectively implemented in the life cycle process. The achievement of software dependability is influenced by

- management policy and technical direction;
- design and implementation processes;
- project specific needs and application environments.

Software organizations are organized and managed groups that have people and facilities with responsibilities, authorities and relationships involving software as part of their routine activities. They exist in governments, public and private corporations, companies, associations and institutions. Software organizations are structured according to specific business needs and application environments for various combinations of development, operation and service provision.

Typical software organizations include those that

a) develop software as their primary product,

b) develop hardware products with embedded software,

c) provide software service support to clients,

d) operate and maintain software networks and systems.

Annex A describes the categorization of software and software applications provided by typical software organizations.

4.3 Relationship between software and hardware dependability

Software behaviour and performance characteristics are different than those experienced in hardware from a dependability perspective. Software codes are created by humans. They are susceptible to human errors, which are influenced by the design environment and organizational culture. Whereas most hardware component failure data are well documented and experienced in use environment, the nature of software faults and their traceability of

[1] Reference in square brackets refers to the bibliography.

cause and effects are not easy to determine in system operation. In most cases the software faults leading to system failures cannot be consistently duplicated. Corrective actions on system failures due to software faults do not guarantee total elimination of the root causes of the software problem.

A bug, after being triggered, results in a software failure (event) and exhibits as a software fault (state). All software faults that cause the inability of the software to accomplish its intended functions are noticed by the software user. Faults and bugs cause problems in the software to perform as designed. Software containing bugs could still accomplish its intended function that is not noticeable to the user. Bugs could cause failures, but could also create nuisance issues that are not affecting a certain function. A software fault can cause system failure, which may exhibit systematic failure symptom.

Software systems and hardware products also have many similarities. They both are managed throughout their design and development stages, and followed by integration and test and production. The discovery of failures and latent faults occur through rigorous analysis, test and verification process with high-levels of test or fault coverage. The high-levels of coverage of the verification process are determined by the assessment of its percentage of fault detection, or fault detection probability. While the management techniques are similar, there are also differences [5, 6]. The following are some examples:

- Software has no physical properties, while hardware does. Software does not wear-out. Failures attributable to software faults appear without advance warning and often provide no indication that they have occurred. Hardware often provides a period of gradual wear-out and possibly graceful degradation until reaching a failed condition.
- Changes to software are flexible and much less time consuming or costly as compared to hardware design changes. Changes to hardware designs require a series of time-consuming adjustments to capital equipment, material procurement, fabrication, assembly, and documentation. However, regression testing of large and complex software programs could be constrained by time and cost limitations.
- Hardware verification and testing is simplified since it is possible to conduct limited testing through knowledge of the physics of the device to analyse and predict behaviour. Software testing can also become simplified through regression testing and analysis to verify minor changes to software due to an identified failure cause. However, minor changes to correct probabilistic failure causes of software, such as race conditions, could lead to very elaborate test and verification cycles to demonstrate adequate correction of the problem.
- Repair and maintenance actions would restore hardware to its operational state generally without design changes. Software repair and maintenance would involve design changes with new service packs or software releases to correct or rectify software faults.

4.4 Software and hardware interaction

Software and hardware interaction occurs in system operation. Dependability issues exist in the interface between the hardware and the operating system. The issues are generally resolved by incorporation of error detection and correction techniques, and exception handling of the hardware and the operating system to mitigate physical faults, and information and timing errors that exist in the interaction. The advent of multi-core processors has enabled redundant multi-threading to enhance dependability in system performance. This enables the user, programmer, or system architect to influence and exploit the redundancy inherent in the multi-core processors to enhance detection and recovery from errors. This also provides opportunity for recovery from soft errors or transient errors that affect either hardware or software or both. The exploitation of increased complexity in multi-core redundancy should be taken into consideration in such applications.

In any control system, the system is controlling some physical processes of actual hardware devices such as sensors and actuators that can fail in system operation. Many of these devices contain embedded software not accessible to the system designer or architect. Examples include smart sensors that contain error detection, redundancy and some error correction features, which are driven by the embedded software. It is important to review the software control algorithms. This is to ensure that the control algorithms are resilient to bad

sensor data and missing sensor values, and that they can detect failed actuation and are capable to compensate or revert to fail-safe condition. Sensor feedback is essential to confirm successful actuation. The feedback mechanism should contain some independent checking of the effects of the commanded actuation. The control system behaviour, assumptions and failure modes should be considered in the design of the software control system.

Intentional and malicious injection of hardware faults to thwart or foil the software algorithms could happen when the system is exposed to deliberate cyber attack. For example, one can inject hardware faults into a cryptographic system to extract the key, or inject a virus into the USB device that is used to initialize a voting machine. The software and hardware interaction could create serious problems to the system operation and affect the dependability in system performance.

Interoperability problems associated with software and hardware interaction could also exist when the software is inappropriately reused in a different environment or for a different application.

The solution to dependability problems related to software and hardware interaction is to increase better understanding of how the new technological system works, and to exercise caution in conducting dependability assessment and testing to fully consider the effects of hardware failures on the software system.

5 Software dependability engineering and application

5.1 System life cycle framework

A system life cycle framework should be established to guide product development and system implementation. The framework is used for defining the system life cycle and governing the performance of the system life cycle processes. IEC 60300-3-15 describes the engineering of system dependability and life cycle implementation, which is based on the technical processes of ISO/IEC 15288 [7]. This applies to any system, whether composed of hardware, software or both.

5.2 Software dependability project implementation

Software engineering activities during the design cycle and useful life period of the system life cycle should be planned, coordinated and managed accordingly along with their hardware counterparts. Engineering activities during the useful life period would involve design changes that could be caused by high failure rates in the customer application, or hardware obsolescence while supplying spares for sustainment of operations. As the hardware changes over the product life cycle, the software would need to change as well. Changes to the software are necessary, as the system design requires forward and backward compatibility between different versions and configurations of the system design.

Dependability activities should be integrated in the respective project plans and incorporated in the system engineering tasks for effective system design, realization, implementation, operation and maintenance. The guidance to engineering dependability into systems per IEC 60300-3-15 applies to this standard. The guidance on software aspects of dependability consists of the following recommended procedures for software dependability achievement in software project implementation:

a) identify the software application objectives and requirements relevant to the software life cycle (see 5.3) and application environment (see Clause A.2);

b) identify the applicable software dependability attributes (see 5.4) relevant to the software project;

c) review the adequacy of dependability management processes and available resources to support software project development and implementation (see 5.5);

d) establish software requirements and dependability objectives (see 5.6, Annex B);

e) classify software faults (see 5.7) and identify relevant software metrics (see 6.2, Annex E) for software dependability strategy implementation (see 5.8);

f) apply relevant dependability methodology for software design and realization (see 6.1, 6.3);

g) initiate dependability improvement where needed taking into consideration of various constraints and limitations for project tailoring (see 6.4, 7.2);

h) monitor development and implementation process for control and feedback to sustain software operability and assure dependability in system operation (see Clause 7).

5.3 Software life cycle activities

The software life cycle encompasses the following activities:

- *requirements definition* identifies the system requirements for combined hardware and software elements in response to the users' needs and constraints of system applications;
- *requirements analysis* determines the feasible design options and transforms the system requirements for service applications into a technical view for hardware and software subsystem design and system development;
- *architectural design* provides a solution to meet system requirements by allocation of system elements into subsystem building blocks to establish a baseline structure for software subsystem decomposition and identify relevant software functions to meet the specified requirements;
- *detailed design* provides a design for each identified function in the system architecture and creates the needed software units and interfaces for the function which can be apportioned to software, hardware, or both. The functions apportioned to software are defined with sufficient details to permit coding and testing. The software function can be labelled as software subsystem and identified as a software configuration item for design control;
- *realization* produces the executable software units that meet verification criteria and design requirements including lower level activities in
  - *coding* of the software units;
  - *unit test* for verification of software unit to meet design requirements;
  - *subsystem test* for verification of software program functions to meet design requirements;
- *integration* assembles the software units and subsystems consistent with the architectural design configuration and installs the complete software system in the host hardware system for testing;
- *acceptance* establishes the system capability and validates the software applications to provide the required performance service for specified system operations in the target environment; software acceptance tests include lower level activities in
  - *reliability growth testing* to increase the reliability of the software system; the testing is conducted after the software system is fully integrated and executed in simulated field operational conditions representing the target environment;
  - *qualification testing* to validate acceptance of the software system for customer release;
- *software operation and maintenance* engages the software in system operation, sustains the system operational capability and responds to application service demands to deliver specific operational services;
- *software update/enhancement* improves the software performance with added features;
- *software disposal* terminates the support of specific software service.

Annex B presents typical software system requirements and related dependability activities for the software life cycle stages.

Figure 1 shows the key dependability activities important to the software life cycle identified for project implementation.

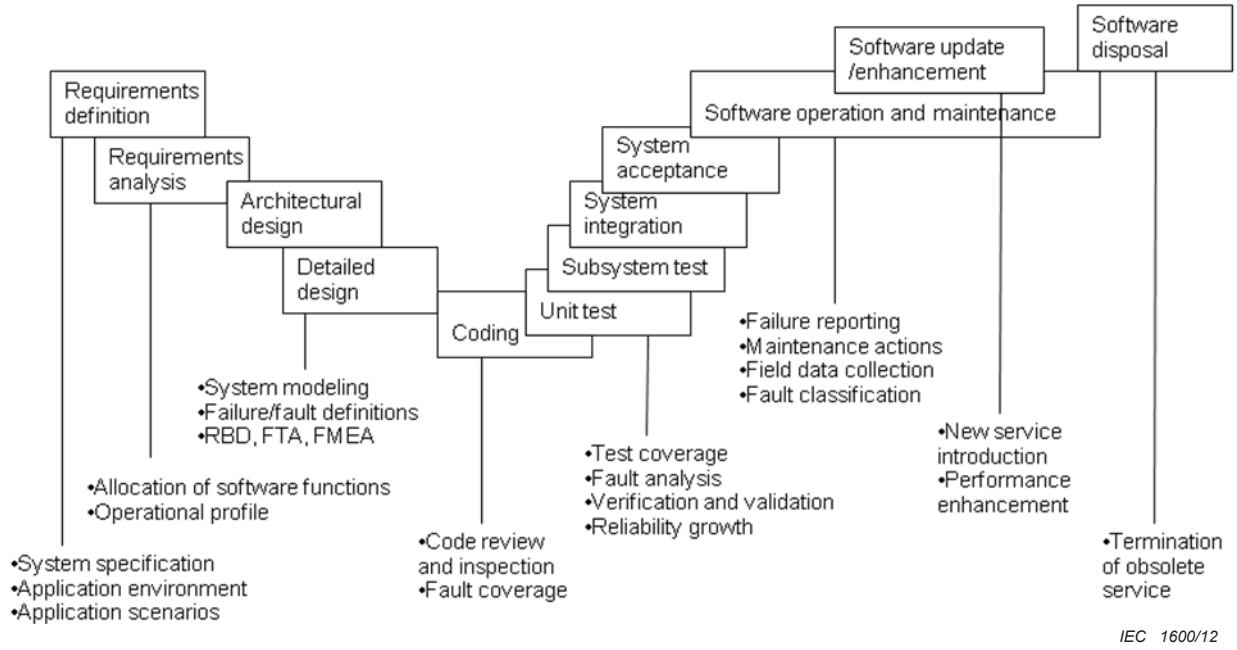

Figure 1 – Software life cycle activities

5.4 Software dependability attributes

Software dependability attributes are those characteristics inherent in the software by design. Specific application related performance attributes should be taken into consideration for incorporation in system design and construction to achieve combined hardware/software system dependability objectives.

The main software dependability attributes or inherent software dependability characteristics contributing to system dependability objectives include:

- *availability*: for readiness of software operation;
- *reliability*: for continuity of software service;
- *maintainability*: for ease of software modification, upgrade and enhancement;
- *recoverability*: for software restoration, following a failure, with or without external actions;
- *integrity*: for correctness of software data.

The specific application related performance attributes contributing to system dependability objectives include, but not limited to the following:

- *security*: for protection from intrusion in software application and use;
- *safety*: for prevention of harm in software application and use;
- *operability*: for robust, fault tolerant, and non-disruptive operation;
- *reusability*: for using an existing software for other applications;
- *supportability*: for sustaining system performance with logistic and maintenance resources;
- *portability*: for cross platform applications.

These inherent software dependability characteristics and specific application related performance attributes form the basis for software system design and application.

5.5 Software design environment

The dependability management objective is to provide a well-balanced design environment for creativity within project budget resources, time schedule and delivery targets. Organizations associated with software development and provision of software services are user application oriented. Tailoring for software projects is needed to manage the allocation of available resources and seek out appropriate design options for effective implementation. The selection and adoption of applicable processes for engineering dependability into a specific software system is accomplished through the project tailoring process for effective dependability management. The recommended tasks for implementation of tailoring process are provided in 7.2. The opportunities for outsourcing design construction, software reuse, and application of commercial-off-the-shelf (COTS) software products for system integration should be explored.

The software design environment relies on an organized process to promote good design practices for error-free code generation, minimize mistakes in requirements definition, and assure test validation for software release. The cultural aspects in software management approach often adopt a capability maturity model concept for infrastructure development [8]. This is similar to the formal implementation of *Capability maturity model integration* [9] described in Annex C for software process management. Software development is a technical process following established software engineering disciplines and application guidelines. The software design environment and practice principles should be included in the organization's policy to establish mission and goals for dependability achievement.

Software design often engages the applications of CASE (Computer-aided software engineering) tools. An effective automated system provides the computational accuracy, traceability of data, configuration management, and a means for collecting the required measurements or input metrics to the models automated. Most data collection systems for field failure reporting, analysis and corrective actions are automated for the same reasons. Historical experience data on software products and services is an indispensable and valuable asset.

5.6 Establishing software requirements and dependability objectives

Software requirements should be established for the software life cycle stages. Applicable dependability activities should be identified for implementation relevant to each stage. Timing for implementation of relevant dependability activities is important. Dependability applications are time dependent and have extensive impact on system life cycle cost [10]. Project tailoring is essential for design trade-offs and constraints resolution. The software requirements and dependability objectives should form part of the overall software product specifications. The strategy for software dependability implementation is described in 5.8. The methodology for engineering dependability into software modules or units and for building software system architecture is addressed in Clause 6. Tailoring process is described in 7.2.

The dependability activities associated with the software requirements are application specific. They reflect the software design and implementation needs to deliver the required system functions for service performance applications. Systematic approaches for implementing relevant dependability activities throughout the software life cycle would ensure the achievement of dependability objectives. Specific dependability objectives are derived from the selection of key dependability attributes and the relevant quantitative metrics. Software dependability requirements can be formulated for specific projects by using the baseline information contained in Annex B.

The influencing conditions on combined hardware/software system dependability specifications are described in the system dependability specifications [11]. The following influencing factors affecting dependability achievement in software development should be considered:

- the organization's design culture, the capability maturity process, and the experience in software design, development and implementation (see Annex C);

- understanding the application environments, user needs, and changing market dynamics for new platform or feature development for practical implementation;
- documentation processes such as failure reporting, data collection, software configuration management for control of software versions and maintenance of experience data records;
- application of software design rules for fault avoidance by controlling the design processes to optimize software performance in software complexity, program complexity, and functional complexity;
- effective use of applicable software methods and tools such as structured design, fault tolerance, design review [12], and software fault management to enhance reliability growth;
- selection of appropriate higher order of programming languages more suitable for specific software structured development;
- established requirements for qualification and measurement of software dependability characteristics.

5.7 Classification of software faults

Software faults could be classified as specification faults, design faults, programming faults, compiler-inserted faults, or faults introduced during software maintenance.

Classification of software faults provides a means for capturing and grouping relevant software fault information. The classification process helps software designers to discover unusual fault patterns for corrective actions. The objective is to eliminate the recurrence of the class of similar faults.

The *orthogonal defect classification* (ODC) [13] is a method used in software engineering for analysis of software fault (defect) data. The ODC addresses the causal effects of quality issues concerning software design and code in a procedural language environment. A defect is a non-fulfilment of a requirement related to an intended or specified use of the software. In this context, a fault due to the inability of the software to perform its required functions exhibits the characteristics of defect attributes in the ODC scheme. Defect attributes are the signature of a defect containing relevant information related to the software fault. The ODC method captures the software fault information of the defect attributes for analysis and modelling. The analysis of ODC data provides a valuable diagnostic method for evaluating the maturity of the software product at various stages of the software life cycle. The ODC can also be used to evaluate the process by analysing the types of triggers to identify specific technical needs to stimulate the missing triggers. The causal analysis of fault (defect) data presents a means for software fault reduction and reliability improvement.

The ODC defect attributes are classified as *Activity*, *Trigger*, *Target*, *Defect Type*, *Defect Qualifier*, *Source*, *Impact*, and *Age*. They are normally collected and analysed during software development to benefit design improvement. The information on defects is available at two specific points in time. When a fault is found, the circumstances leading to the fault exposure and the likely impact to the user are generally known. When a fault is closed after the fix is made, the exact nature of the fault and the scope of the fix are known. ODC categories capture the semantics of a fault (defect) from these two perspectives. By defining the *Activities* during the development process and their mapping to the ODC *Triggers*, the ODC provides valuable insights and customized the fault (defect) information for the software development organization.

The set of defect attributes is summarized: a) when a fault is found which is known as *opener section*, and b) when a fault is fixed which is known as *closer section*. ODC is most useful in mature software organizations where extensive data are normally collected and analysed for software product improvement.

Annex D presents a summary of classification of software defect attributes.

5.8 Strategy for software dependability implementation

5.8.1 Software fault avoidance

Software codes are generated to produce a software product. A mistake made during software design and coding could manifest itself when triggered, to become a software fault leading to a system failure. Since faults are the main cause of system failures, preventing faults from being introduced during design, such as code review and removing the residual faults that had escaped detection by testing, are common approaches to lessen the existence of fault problems for the software life cycle. The recommended software fault avoidance strategy includes fault prevention and fault removal.

a) Fault prevention
   - Establish fault prevention objectives in software engineering disciplines.
   - Initiate requirement specifications review.
   - Conduct early user interaction and refinement of the software requirements.
   - Introduce formal methods where applicable and practicable.
   - Implement systematic techniques for software reuse and assurance for application.

b) Fault removal
   - Detect and eliminate the existence of software faults by testing.
   - Conduct formal inspection on finding faults, correcting faults, and verifying the corrections.
   - Perform corrective and perfective maintenance actions during software in-service operation.

5.8.2 Software fault control

Software faults are difficult to detect and fault removal can be achieved by various means including rigorous software testing and inspection. Exhaustive testing of software is often limited by time and cost constraints in project management. Dependability assurance based on testing alone does not guarantee complete fault elimination. Software fault control employs fault tolerance and forecasting methods to minimize the manifestation of latent software faults or bugs that can still exist after the software release for use. The recommended software fault control strategy includes fault tolerance and fault/failure forecasting.

a) Fault tolerance
   - Establish methodology for fault confinement, fault detection and fault recovery.
   - Implement software design diversity and fall-back schemes.
   - Introduce multi-version programming techniques.
   - Implement self-checking programming techniques.

b) Fault/failure forecasting
   - Establish fault/failure relationships in operational environment.
   - Establish data collection system to capture relevant data.
   - Conduct reliability growth testing where applicable.
   - Develop and implement relevant reliability models for fault/failure estimation.
   - Refine forecasting techniques for time projection of software version release.

6 Methodology for software dependability applications

6.1 Software development practices for dependability achievement

The capability maturity in software development reflects an organization's ability to develop software with consistency and dependable products for intended applications. The following fault avoidance and fault control techniques are recommended for incorporation where applicable in software development:

a) standardize methods for high-level architectural design, detailed design, coding and testing, and documentation to facilitate communications and fault avoidance;

b) develop modular designs for software units and subsystems with well-defined software functions and interfaces by building simple, separate and independent software units to facilitate design interaction, maintenance, error traceability, fault mitigation and bug removal;

c) use design patterns, which are general reusable solutions of well-tested software, as templates for solving software design problems to speed up the development process;

d) institute formal design methods where appropriate for control and documentation of software design and development process;

e) utilize *software reliability engineering* [14] techniques for software reliability assessment and enhancement [15];

f) reuse software available from software library on well-tested software units and subsystems for similar application and operational profile to reduce development cost and time, and minimize new design fault introduction;

g) develop regression testing methods to ensure functionality of existing software as new functionality is introduced or fault removal is performed;

h) testing software units and subsystems to verify low-level design functions and validate integrated high-level design architectural system performance for progressive bug removal to prevent fault propagation;

i) conduct inspections and reviews of software design requirements, software codes, user manuals, training materials, and test documents to detect and eliminate as much as possible mistakes; different review teams for comparison of results should be considered and employed where practicable;

j) control change to reduce fault occurrences such as version and change control process in software configuration management;

k) analyse root cause problems and implement appropriate corrective actions for continuous software improvement;

l) establish data collection system for knowledge base capture of software faults and performance data history.

6.2 Software dependability metrics and data collection

Software dependability metrics are measures of dependability characteristics of a software system. The measurement of metrics provides a quantitative scale and method to determine the value of a specific characteristic associated with the software system. These industry standard metrics are obtained either by direct measurement or by deduction. They are used for software system performance measurements. The following software metrics are de facto industry standards for application. They should be considered where appropriate for software system dependability assessment.

a) *Availability*: provides a measure of up time over the duration of system operation.

b) *Failure frequency*: provides a measure of the number of failure over the duration of system operation.

c) *Time-to-failure*: provides a measure of the failure-free time period.

d) *Restoration time*: provides a measure of the time for restoration of a system from a failed condition (down state) back to normal operation (up state).

e) *Fault density*: provides a measure of the number of faults contained per kilo source lines of code (KSLOC) or per function point and is used for software reliability assessment.

f) *Function point*: provides a measure of the functional size of application software for software project planning by means of *function point analysis* method [16].

g) *Code coverage*: provides a measure of the degree to which the source code and the logical branches of a software program that has been systematically tested; *code coverage* is an indicator on the thoroughness of software testing, it is used to represent *fault coverage* that indicates the percentage of faults detected during the test in code execution.

h) *Fault removal rate*: provides a measure of the number of faults detected and corrected in a software product for a defined period of time or software execution duration; *fault removal rate* is used in reliability growth to establish reliability improvement trend.

i) *Residual faults in software*: provide a measure of the estimated number of bugs still remaining in the software product after testing for bug removal.

j) *Time for software release*: provides a measure of the estimated time for software product release schedule based on established criteria on an acceptable level of bugs still remaining in the software product for software project management.

k) *Software complexity*: provides a measure of the degree of difficulty for design and implementation of a software function or a software system; other complexity measures based on the complexity concept include program complexity, functional complexity, operational complexity; complexity-related metrics are used as inputs for reliability assessments and prediction models.

There are numerous metrics used for various reasons during the software life cycle. Software metrics can be grouped into three general categories to facilitate data collection.

1) *Fault data metrics*: capture the software problem reporting data for measuring the impact of the faults and the efficiency of the reporting process to improve software maintenance.

2) *Product data metrics*: capture the software product information by categorizing the size, functionality, complexity, location of use, and other characteristics to facilitate the experienced data as inputs to benefit new product development. The metrics provide performance history and data, and information of various software product groups.

3) *Process data metrics*: capture the software restoration process information and conditions at the time of fault detection and removal for reliability model inputs in reliability prediction.

The data collection process is critical for measuring software dependability attributes and performance characteristics. An effective data collection system should be practical for implementation. The amount and types of data should be relatively simple to collect, easy to interpret for data analysis, and useful for software dependability assessment, improvement and enhancement. The data collected is used to determine system reliability trends, frequency and time duration needed for software maintenance, response time for service calls, degraded performance restoration and maintenance support requirements.

Annex E presents examples of software data metrics obtained from data collection.

6.3 Software dependability assessment

6.3.1 Software dependability assessment process

The objective of software dependability process implementation is to ensure software system maturity in development and dependability achievement. The assessment process is the enabling mechanism to ensure verification of software requirements and validation of software dependability in system performance results. The dependability assessment process incorporates crucial software engineering activities adopted from established industry

practices [14]. The following process for conducting software dependability assessment is recommended:

- identify user needs and system performance objective and develop dependability specification;
- establish software operational profile;
- allocate applicable dependability attributes;
- perform dependability analysis and evaluation to determine options and possible solutions;
- conduct software testing and measurements;
- conduct software verification and software system validation;
- perform software reliability growth and forecast improvement trends;
- evaluate assessment results and feedback.

The software dependability assessment activities are described in the following sub-clauses.

6.3.2 System performance and dependability specification

The purpose is to identify the system performance objective for development of a system dependability specification [11] if not provided by the customer or user. The following process is recommended:

- identify the system performance scenario and application environment;
- identify the relevant performance influencing factors;
- identify the system boundary and interfaces with external interacting systems;
- identify the relevant system performance attributes;
- identify the system architecture, hardware/software configuration;
- identify the interoperating hardware and software functions of the system configuration;
- characterize and quantify the dependability attributes of the relevant hardware and software functions; including availability, reliability, and recoverability associated with maintenance support criteria.

Documentation of the system dependability specification should include the following data as part of the system specification:

- system identification;
- system performance objective;
- system operational profile;
- system dependability performance targets;
- system configuration;
- system functions;
- dependability requirements for each function;
- system maintenance support requirements.

For software system, it is essential to consider the operational profile that affects the execution time of software functions for on demand applications. A functional *reliability block diagram* [17] can be constructed to represent the hardware/software system. The functional blocks created would facilitate allocation of the respective software dependability metrics to each software function established according to the software system architecture and software configuration.

6.3.3 Establishing software operational profile

An operational profile is the sequence of required activities to be performed by the combined hardware/software system to achieve its mission or service objective. The system performance is highly dependent on the environment in which the system operates. The environment can affect hardware physical changes, but cannot affect the software functions delivered by the execution of software programs in system operation.

The development of an operational profile is a quantitative characterization on how the software is being used. Operational data and relevant information are usually gathered through customer surveys and gained by field service experiences. The following are recommended processes for development of an operational profile:

a) determine the customer profile by establishing the needs and types of customers, such as an organization or an individual intended to acquire or purchase the software system;

b) establish the user profile on the different types of users, such as a person or an employee of an organization, or interacting software application systems, operating or using the software system for specific applications;

c) define the system-mode profile on how the system is being operated and in what sequence or order expressed in terms of modes of operation, such as software testing for upgrade maintenance, or normal batch data processing in executing the software system;

d) determine the functional profile by evaluation of each system mode for performance functions and service features, such as create e-mail message or address look-up in meeting the software functional requirements;

e) determine the operational profile based on the functional profiles established for system performance functions;

f) determine the information profile by collecting software application data at software development life cycle.

The functional profile is a user-oriented view of system capabilities. From the developer’s perspective, functional profile represents the system operations that actually implement the required functions. From a dependability perspective, the operational profile is a set of different system operating scenarios and their probabilities of occurrence. The operational profile provides the needed inputs for development of test cases to simulate software system field operations and specific applications of the software functional features usage. The execution of test cases for software testing provides valuable information and data capture for estimation of software reliability in field operation and validate the provision of maintenance functions and efficiency of maintenance support performance.

6.3.4 Allocation of dependability attributes

Allocation of dependability attributes and measures for software system is based on the concept of modelling system architectural functions to reflect the requirements of the system dependability objective. The initial value assignment of applicable dependability metrics such as reliability and availability, are most likely based on experience data. These metric values are further refined through iterative analysis and evaluation process. The apportionment of reliability and availability values to the various software subsystems and functional units are assigned according to their complexity, criticality, estimated achievable reliability or availability performance targets, and other influencing factors relevant to the allocation process.

The development of a system model for software differs significantly from hardware due to its inherent operating characteristics. For each mode in system operation that involves the software program functions as configuration items, different set of constituent software units are being executed. Each mode has a unique time of application associated with the software unit execution duration on demand in system operation. This indicates the time duration of each system mode. The software system modelling includes the number of lines of source code in each software unit, the code complexity and other information pertaining to software development resources, such as programming language and design environment. They are

used to establish the initial failure rate for reliability or availability prediction of the software configuration items.

6.3.5 Dependability analysis and evaluation

The following dependability analysis and evaluation activities are needed to support software system development. The process is iterative for optimization of dependability design requirements to meet system performance objective. Availability/reliability modelling is used for analysis and evaluation of the software time-dependent performance functions.

a) Modelling availability/reliability functions

A simple approach to analyzing the availability or reliability of a system comprised of hardware and software is to form a structural model of the system. A functional availability/reliability model for the combined hardware/software system consisting of functional blocks can be constructed using the *reliability block diagram* (RBD) technique [17]. The model is decomposed into separate subsystem models representing the constituent hardware and software elements of the system. fault tree analysis (FTA) [18], Markov chains [19] and Petri nets [20] are also useful for system availability/reliability model development. For example, FTA can be effectively used to model system reliability with dynamic gates to determine hardware and software availability/reliability functions for trade-offs and improvement [21]. It should be noted that RBD and FTA are logically equivalent. RBD focuses on success; FTA on failure.

The hardware subsystem availability/reliability model consists of all hardware elements of the system with the availability/reliability functional blocks constructed as appropriate to represent the relevant hardware subsystem structure and redundancy configuration. This is to facilitate prediction of failure rates of individual hardware components and for hardware subsystem availability/reliability determination according to prediction techniques. There can be one or more hardware subsystems servicing different functions in the system configuration.

The software subsystem reliability model is constructed using software units as building blocks to deliver software program functions. A software unit is the lowest level of a configurable software item. Software units do not fail independently as with the hardware components. Software codes are virtual entities not subject to physical changes. Software units fail in association with the system operational profile, which affects the configuration scheme of the software reliability model structure. Modelling software reliability needs to incorporate the operational profile information in developing the software configuration structure. The software subsystem program can consist of one or more constituent software units to deliver the required functions. A software subsystem program residing in a host hardware subsystem is configured to form a software configuration item. The interoperation and mutual dependency of the software subsystem and its designated hardware host are needed to deliver specific subsystem software functions for system operation. There can be several combined software and hardware subsystems servicing different functions in the entire system configuration. Annex F presents an example to illustrate the interactions of combined hardware/software reliability functions to derive the system failure rate for reliability assessment.

Availability assessment of combined hardware/software system should first establish the interactions of combined hardware/software reliability functions prior to derivation of system availability functions. The total system downtime, or time for restoration for the duration of system operation, is required for system availability assessment.

b) Determining reliability of software functions

Software failures can occur during system operation. Determining the software failure rates for use in reliability modelling requires that the software be treated as a subsystem, which resides in a host hardware subsystem, configured to form a software configuration item. The software subsystem can perform one or more of its required functions. A function is a capability of the system to deliver a required service from the end user’s perspective. The function can be accomplished by a software configuration item, so that the required service

function is recognized by version control. A software unit designed to perform exactly one single function can be a configuration item. A software subsystem program requiring multiple software units to perform a single function can also be a configuration item. A software subsystem can consist of several software programs to deliver a set of related functions. Each of these software programs is a configuration item by definition. The concept of software configuration item is viewed from a software design version control perspective. Software configuration item is essential for tracking design changes. Each design change is assigned a version issue for identification. Referencing software version is necessary for tracking effectiveness of software maintenance upgrades. Reliability growth trend is established by performance improvement indication with the system running the new version replacing old version. Delivering the required functions in system operation is the challenge to meet system reliability objective.

The software functions that comprise a system are related in a timing configuration and a reliability topology.

Timing configuration is a concern when the various functions are active or inactive during a specific time period in system operation. The major timing relationships among software functions are concurrent or sequential. Functions are concurrent if they are active simultaneously. The functions are sequential if they are active one after the other. It is also possible for function times to partially overlap, resulting in a hybrid concurrent/sequential timing configuration. Concurrent active software functions are found in systems that are serviced by more than one central processing unit, such as in a multiprocessing system or a distributed system. Run time references are identified as execution time, system operating time, and calendar time.

Reliability topology concerns the number of functions in the system that can fail before the system fails. Reliability topology is the relationship of an individual function failure to the failure of the aggregate system. Software functions are generally related in a series topology. The failure of one function would cause in the failure of the software system. Software fault tolerant design can be used to protect a system in the event of failure of one or more functions.

c) Run time reference of software function

Software failure rate can be expressed with respect to three different time frames of reference.

- *Execution time* is the central processing unit run time, which accumulates when the software program is executing instructions. The execution time is used to determine the execution-time failure rate of the application software subsystem.
- *System operating time* increments whenever the hardware/software system as a whole is operating. This is used to determine the operation-time failure rate of continuous operation software subsystem.
- *Calendar time* is the time period used for project planning and scheduling purpose. Calendar time is always incrementing.

During system operation, software programs do not always run continuously. Some programs can time-share a single central processing unit. Multiple central processing units can also be present, allowing the program executions to overlap. The failure rates of the various programs need to be combined to arrive at an overall software failure rate. They are converted into a common time frame of reference in system operating time. This is the same time frame used to express hardware failure rates to facilitate failure rate determination of combined hardware/software system.

d) Criticality of software function

Software functions are often used for control of a critical system where a failure can cause catastrophic consequences. The criticality of software functions should be identified early in system concept definition and evaluated during software architectural design of the system.

The criticality of functional failures should be classified in the system specifications, such as critical, major, or minor based on established criteria; and verified by analysis in system reliability performance.

The level of risk associated with the critical software function can be determined and evaluated by means of risk assessment techniques. Project risk management [22] should focus on fault prevention and fault tolerance where the severity of failure consequences can be mitigated.

Fault tree analysis [18] can be used to identify the possible causes of an unwanted top event. It is used to investigate the potential faults and their causes, and quantify their contribution to system unavailability. Fault tree analysis is a top-down technical approach, where the starting point is from the top-level software subsystem program and following it through the software hierarchical structure to the lowest software unit. The potential faults can be individually identified and assessed on their respective probability of failure occurrences. The quantitative assessment provides an indication or magnitude of the criticality of the software function. This is of interest for design optimization and fault avoidance.

Failure mode and effects analysis [23] can be used to determine possible failure modes and faults in the software units and their effects on the next higher-level subsystem of the software hierarchical structure. Failure mode and effects analysis is a bottom-up technical approach. It can be extended and used for criticality analysis of software functions. The criticality analysis combines quantitative value of the likelihood of failure occurrence and qualitative information on failure severity to support design trade-off and fault mitigation.

Other system dependability analysis techniques [24] are used for software decomposition and system simulation. They can be selectively used for detailed reliability and maintainability assessment of software functions in combined hardware/software systems.

Software integrity level is a value representing project-unique characteristics that define the importance of the software to the user. Examples of project-unique characteristics include software complexity, criticality, risk, safety level, security level, desired performance, and reliability. The software integrity level is determined by classification of criticality of the impact of failure consequences and their associated frequency of occurrences [25]. The criticality of software functions is also application specific. For safety-related systems, the safety-integrity level should be defined and incorporated for software system development to meet functional safety requirements [26]. For security-related systems, specific system security requirements [27] should be incorporated.

6.3.6 Software verification and software system validation

Specification of software tends to be much more complex than specifying physical hardware systems such as machinery and electric/electronic systems. The "correctness" of software is of primary concern. The verification process [2] is to determine that the requirements for the software are complete and correct as applicable to the software life cycle stages. The validation process [7] is to determine that the software system performance and services are conformed to the customer/user requirements. Appropriate enabling systems, such as test equipment, facilities and supplementary resources, are required to support the implementation of the verification and validation processes. The enabling system does not contribute directly to the performance functions of the software or the system under test during operation of its life cycle stages.

a) Software verification

The software verification process is to confirm that the specified requirements are fulfilled by the software system. The following verification process activities are recommended:

- define strategy for software verification;
- develop a verification plan based on software system requirements;

- identify the constraints and limitations associated with the design decisions;
- ensure that the enabling system for verification is available and associated facilities and testing resources are prepared;
- conduct the verification to demonstrate compliance to the specified design requirements;
- document the verification results and data;
- analyse the verification data for initiation of corrective action.

b) Software system validation

The software system validation process is to provide objective evidence that the system performance meets customer/user requirements. The following validation process activities are recommended:

- define strategy for validation of the services in the operational environment and achieving customer/user satisfaction;
- prepare a validation plan;
- ensure that the enabling system for validation is available and associated facilities and testing resources are prepared;
- conduct validation to demonstrate conformance of services to the customer/user requirements;
- document the validation results and data;
- analyse, record and report validation data according to the criteria defined in the validation strategy.

6.3.7 Software testing and measurement

a) General consideration for testing software

Software testing is the process of executing a program or a set of coded instructions with the intent of verifying software functions and finding errors. The software testing objectives vary with project needs, software product availability, software maturity status, and scheduling for testing during the software life cycle. When planning for software testing the following should be taken into consideration.

- Test planning is essential and should be documented to describe the test objectives, test process, procedures and resources.
- Software testing requires knowledge and skills and good testing practice. Although many of the test routines and tools have been automated and widely deployed in industry, good testing techniques demand the skills, experience, intuition and creativity of the tester to achieve dependable results. Maintaining test record is important to provide accuracy and traceability of test data.
- Testing is more than just debugging the software program to locate faults and correct errors. Testing is also used in software verification and validation, availability and reliability measurement.
- Test efficiency and process effectiveness are criteria for coverage-based testing techniques. Test automation can expedite software test time and reduce project cost. The selection of appropriate test tools, the training and support costs associated with the test tool acquisition should be taken into consideration.
- Testing may not be necessarily the most effective means to improve software quality unless appropriate follow-up actions are taken. Alternative methods, such as code inspection and code review should be considered.
- Software testing is only part of the software reliability growth and improvement process. It needs collaboration of other assurance efforts to achieve dependability goals.

- Complete testing may not be feasible or practical, and often time/cost prohibitive. Software complexity influences the extent of test completeness. The complexity problem often limits the tester's ability to detect and remove bugs by the testing process.
- Latent software faults do exist in software after its release for use operation. Software reliability prediction provides a means to estimate the test time required on reducing the residual software bugs to an acceptable number before the software version release.
- Testing beyond unit testing should be performed by testing teams that are separate and independent to the teams developing the software.

b) Types of software tests

The following presents the types of software tests performed during the software life cycle.

- *Unit test*: testing of one software unit that can be compiled before it is integrated into the software program or subsystem. The software unit is tested to verify that the detailed design specifications for the unit has been correctly implemented.
- *Subsystem test*: testing of a subsystem software program consisting of one or more software units as a software configuration item to verify functional performance requirements.
- *Integration test*: testing of a software system in a hardware host as a whole consisting of integrated subsystems to verify functional operation, expose problems in software interfaces, hardware interfaces and interactions between the hardware and software, and validate reliability performance.
- *Reliability growth test*: testing of software in an iterative process to improve reliability through testing until failure, analysing failures, implementing corrective action on the existing software version for upgrade, and continuing the test with the new software version. Termination of reliability growth test is based on when the established software reliability target is met.
- *Qualification test*: testing to demonstrate that the software meets its specifications when integrated in its host hardware system and ready for use in its target environment. Before release of final version for software distribution, alpha and beta testing are often conducted for quality assurance purposes. *Alpha testing* is an in-house trial carried out by software developer before release for external users. *Beta testing* is a field trial carried out by a limited number of users in its intended application to seek user feedback experience information.
- *Acceptance test*: testing of a software system to validate that the customer's requirements are met. For acceptance testing of complex hardware/software systems where no prior information exists on similar systems, reliability growth and stress testing [28] should be considered as part of the acceptance test requirements.
- *Regression test*: testing of software that has been previously tested in an effort to uncover any maintenance errors introduced, new code being developed, improper configuration, or inadequate source control.

c) Testability of software

Testability is the ability of the software to be tested with minimum time and resources. Testability is a design characteristic that allows the software operational status to be determined effectively. The testability design characteristic also permits the process for faults detection, isolation and diagnosis to be performed efficiently. Design for testability should focus on structured design of the software function to enable testing. Modular design approach where each software function is independent of the other functions would facilitate testability in detection and isolation of faults. The approach would enhance maintainability of the software function by simplifying the process for software update or modification.

Self-test programmes, monitoring and control procedures can be designed and incorporated into a software system to perform self-testing of the system. The self-test function can be operated on demand or activated automatically by the programmes to facilitate maintenance and diagnosis of the system indicating its operational performance status. The self-test design

features should include the capability of false alarm detection and indication such as operator error and transient condition. False alarm is a warning reported by the self-test diagnostic management function indicating the existence of an operational fault when that fault does not exist. False alarms can be reduced or eliminated by a full and accurate diagnostic analysis and validated by the run-time diagnostic management process.

The structured design approach for testability involves a process for rationalization of the objectives for testing. The process analyses the software attributes and predicts the likelihood if there are any bugs in the software that can be revealed through testing. The analysis is used to optimize the testing process to determine how much testing is enough. It provides the means to manage test resources and determine the value or benefits of a specific testing approach.

d) Test cases

Test cases are developed based on the software specifications. Test cases are used to simulate actual software field operating conditions in which specific interest areas or potential problems could be encountered. A test case is a set of test inputs, execution conditions, and expected results developed for a particular testing objective; such as to exercise a particular software program path or to verify compliance with a specific requirement. A test case specification is the documentation for specifying inputs, identifying expected test results, and establishing execution conditions for the test item. An effective testing process includes both manually and automatically produced test cases. Manual tests cover the depth of finding software faults reflecting the developer's understanding of the problem domain and data structure. Automatic tests cover the breadth of fault investigations by executing the entire range of test values, including those extremes that manual tests might miss. The automatic test process engages the use of a test case generator to accept source code, test criteria, specifications, or data structure definitions as inputs to generate test data and determine expected results. Fault insertion test could be considered as one of the test cases in which a deliberate fault is introduced in one part of the software system to verify that another part reacts appropriately. The test results are used to determine probable fault conditions and facilitate software fault-tolerant design. Fault insertion technique is also used to test the coverage of the test program by counting the fraction of the inserted faults found.

Testing a software program is an attempt to make the software fail. It is important to note that any failed execution must yield a test case for inclusion in the software project's test suite. The most important aspect of a testing strategy is the number of faults that the test has uncovered as a function of time. This provides an indication of test efficiency.

e) Software measurement and metrics for project management

Measurement is the process of determination or estimation of quantitative values or metrics to facilitate effective project management. The metric data are obtained by various methods described as follows from different perspectives for software reliability prediction and system dependability performance improvement.

- *Design structure metrics*: measurement of design approach, complexity, and independence of the software design.
- *Design completion metrics*: measurement of the extent to which the software performs the specified functions completely.
- *Process management metrics*: measurement of the management, cost-effectiveness, and design trade-offs for the software based on the analysis results of the process data metrics and the relevant fault data metrics captured in the data collection process.
- *Product management metrics*: measurement of the characteristics of the software that are specific to the software product developed based on the analysis results of the product data metrics and the relevant fault data metrics captured in the data collection process.

Many of these metrics are used for inputs to software reliability model parameters for prediction and estimation where quantitative values are needed.

Annex G presents a summary of software reliability model metrics commonly used in industry practice.

6.3.8 Software reliability growth and forecasting

Software reliability growth is the condition characterized by a progressive improvement of a reliability performance measure of the software system with time. Software reliability improvement is achieved by design and the progressive reliability attainment is verified by means of reliability growth testing. Software does not fail if it is not executed to expose failures. A software program can only fail when it is executed. The software failures uncover faults, and the removal of these faults results in reliability improvement. Software reliability growth trends are based on the fault removal rates with respect to the cumulative software execution time. For scheduling purposes, execution time can be converted to calendar time to establish the software failure rates for reliability estimation. A reliability growth program [29] can be established for combined hardware/software system. The reliability growth models and estimation methods for assessments, based on failure data captured in the reliability growth program, are described in the statistical methods for reliability growth [30]. Typical software design improvement methods are provided in 6.4. Software reliability growth testing is presented as follows.

a) Software reliability growth testing

Reliability growth testing is performed to assess current reliability, identify and eliminate bugs, and forecast future reliability. The reliability values based on the bug counts uncovered and removed during the execution time period are compared with intermediate software reliability objectives. This is to measure reliability progress trends in the testing process to achieve software reliability targets.

Accelerated testing [31] has been successfully used to shorten product test time. This is accomplished through the application of increased stress levels or by increasing the speed of application of repetitive stresses to assess or demonstrate product reliability growth. For combined hardware/software systems, the stress application to software can involve test excursions through various operational scenarios. The objective is to verify system performance adequacy under simulated operating environments within practical test time and conditions.

b) Software execution environment

The software execution environment includes the hardware platform such as the hardware host system, the operating system software, the system generation parameters, the workload, and the operational profile. The operational profile is described in 6.3.3.

A *run* is the result of execution of a software program. A run has identifiable input and output variables. Software reliability testing is based on selecting a set of input variable values for a particular run. Each input variable has a declared data type representing a range and ordering of permissible values. An operational profile is a function associated with the probability of the input variable, which is used in statistical estimation for reliability growth.

c) Multiple software copies

The time on test during reliability growth testing can be accumulated on more than one copy of the software. The copies can run simultaneously to accelerate testing. This procedure permits multiple copies run time accumulation to speed up the testing process, especially to demonstrate achievement of high reliability targets. In this respect, the total amount of calendar time on test is reduced. The use of multiple copies can provide economic and scheduling advantages.

d) Software reliability forecasting

Reliability growth for software is the positive improvement of software reliability over time, accomplished through systematic removal of software bugs. The rate of reliability growth depends on how fast the bugs can be uncovered and removed. A software reliability model applicable to growth conditions allows project management to track the software reliability progress through statistical inference to establish trends and forecast future reliability targets. Appropriate management actions can also be taken if the trend indicates negative.

e) Software reliability models

Measuring and projecting software reliability growth requires the use of an appropriate software reliability model that describes the variation of software reliability with time. The parameters of the reliability model can be obtained either from prediction based experience data, or from estimation of test data collected during system test. The selection and use of software reliability model should be validated. The estimation process is based on the times at which the failures occur with sufficient data sample for significant execution time accumulation. This is to establish a reasonable degree of statistical confidence to validate the reliability growth trends. The approach is to forecast software maturity and release targets.

Annex H presents typical examples of software reliability models used in industry practice.

6.3.9 Software dependability information feedback

Software dependability data collection is addressed in 6.2. The data collection activity is conducted for field tracking to assess the dependability of software performance operation in customer premises. This is to ensure and confirm that the accepted level of dependability performance in operation is sustained for the software deployment. The in-service field dependability information is collected together with relevant customer feedback information. The information is used to justify changes for new software requirements and initiate development of software new release.

Often times due to the dynamics of application environments and technology evolution, the decisions on software new releases are influenced by market competitions and driven by business strategies.

6.4 Software dependability improvement

6.4.1 Overview of software dependability improvement

Software dependability improvement can be achieved by improvement in software design, improvement through reliability growth testing, and improvement in software maintenance support performance for customer support services, including software enhancement effort.

Reliability is a key attribute of software dependability. Software reliability improvement through reliability growth testing is described in 6.3.8. The following subclauses provide practical approaches relevant to software reliability design and recommended techniques for software enhancement and implementation. The design objectives are focused on *testability* for ease of verification of software functions, *modularity* for independence of each software function to facilitate fault isolation and containment, and *maintainability* for ease of modification in software life cycle.

6.4.2 Software complexity simplification

a) Structural complexity

Structural complexity describes the logic paths for software module connection of software design. Each module unit could be programmed (by coding) to provide an executable unit of software function in the software structure. Structural complexity is related to the testability of program codes that affect fault detection, hence influencing the reliability and maintainability of the software architecture. The more complex the structure, the harder it is to test the software. The software design rules should establish a level of complexity to facilitate design for dependability.

b) Functional complexity

Functional complexity describes the required functions that software module or segment of code in the unit must perform. Ideally, one module unit should be designed to perform one function to achieve simplicity with one set of cohesive inputs and outputs to facilitate software fault isolation and removal. In practice, both structural complexity and functional complexity should be considered for software design evaluation. Software design strategy on complexity is directly linked to the number of test cases needed for complete software verification.

6.4.3 Software fault tolerance

Fault tolerance is the software ability to continue functioning and preserve the integrity of data with certain faults present. Software fault tolerance design is to prevent software faults from causing system failure during system operation. Software fault tolerance is constructed to have a low probability of exhibiting common-mode failure from a number of diverse system designs including the following recommended practices.

- *Fault confinement*: software is written in such a way that when a fault occurs, it cannot contaminate portions of the software beyond the local domain where it occurred.
- *Fault detection*: software is written such that it tests for and responds to faults when they arise.
- *Fault recovery*: software is written that after detecting a fault, it takes sufficient steps to allow the software to continue to function successfully.
- *Design diversity*: software and its data are created so that there are fall-back versions available.

Fault tolerance exhibits the graceful degradation property that enables a software system to continue operating properly for a period of time in the event of failures. This is to prevent failures that would otherwise cause abrupt system outage or total breakdown. Fault tolerance is of particular importance for safety critical systems that depend on high availability system performance in the presence of faults or operating under adverse conditions. An example of fault tolerance design is the Transmission Control Protocol for Internet communications. This is a software protocol designed to allow reliable two-way communications in a packet switch network, even in the presence of communication links that are imperfect or overloaded. The fault tolerance design is accomplished by requiring the end points of the communication to expect packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount to sustain operation.

Multi-version programming method is a possible approach used for fault tolerance in design of critical systems and for improvement of software reliability in operation. The method engages multiple functionally equivalent programs that are independently generated from the same initial software specifications. The independence of separate programming effort would greatly reduce the probability of identical software faults occurring in two or more versions of the program. Implementation of these programs utilizes different algorithms and programming language. Special mechanisms are built into the software to allow these separate programs to be controlled by a voting scheme in the decision algorithm for program execution in application. The concept is based on the assumption that output from multiple independent versions is more likely to be correct than the output from a single version from a redundancy view-point. In practice, the improvement benefits of multi-programming effort would require justification of the additional time and resource requirements to warrant cost-effective implementation. The effectiveness of the method would also depend on assuring diverse fault characteristics between versions in software design and implementation.

6.4.4 Software interoperability

Software interoperability is the ability of diverse software systems to work together to exchange information and to use the information that has been exchanged. In an open system such as an IP network, it is important to achieve interoperability of diverse software systems to establish communication links. Failure of the communication link would affect dependability

in performance operation. One practical approach recommended to enhance interoperability in communication network is to incorporate a specific feature in the software system design to monitor the situation of the established communication. For example the “heartbeat” technology (signal processing and synchronization scheme) incorporating the monitoring feature is to send the “heartbeat” signal to each other when the communication link is established. If the link is broken or interrupted due to changes in the environment or any other causes, the software system will automatically attempt to seek and re-establish the link to maintain continued communication such that dependability in performance operation of the communication network is not degraded.

6.4.5 Software reuse

Software reuse is motivated by various reasons including proven history in performance operation, economy in time and cost savings, and proprietary products in business decisions.

Software reuse is the use of existing software to build new software. Reusable software is a reusable asset. The most well known reusable asset is code. Programming code written at one time can be used in another program written at a later time. The reuse of programming code is a common technique that attempts to save time and effort by reducing the amount of repeated work. Software *reusability* is the degree to which a software asset can be used in a different software system or in building other assets.

From a dependability perspective, the application of software reuse in projects and its *reusability* attributes should be controlled to achieve dependability improvement. Reusability is directly dependent on the software structure and modular design. For a software unit to be reusable it should be confined to perform only one function completely. This restriction is essential because if the intended reuse of the software unit is performing less than one function, or it is able to perform more than one function, it would be difficult to implement or to maintain for its intended reuse purpose. Deviation from such restriction would decrease the usefulness of the reusable software. Reusing software that does not perform exactly one function could have adverse effect on dependability due to the possibility of errors introduced into the software during implementation or maintenance.

The reuse of software should be implemented only if the functional requirements of the new software unit are in line with those of the reusable software for very similar application and operational environment. Otherwise, it would diminish the cost-effectiveness of the software reuse objective and possibly decrease reliability when implemented.

Reusable software should be well documented for traceability to facilitate configuration management of software assets. Commercial off-the-shelf (COTS) software products and systems should be treated as reusable software for varied multiple applications. Qualification testing for assurance purposes should be implemented to validate COTS software product/system performance and suitability to meet project application needs.

6.4.6 Software maintenance and enhancement

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other software performance attributes, or to adapt the product to a modified environment. There are four main categories of software maintenance.

- *Corrective maintenance*: reactive modification of a software product performed after delivery to correct discovered problems.
- *Adaptive maintenance*: modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- *Perfective maintenance*: modification of a software product after delivery to improve performance or maintainability.
- *Preventive maintenance*: modification of a software product after delivery to detect and correct bugs in the software product before allowing further propagation into real failure occurrences.

The key software maintenance issues are both managerial and technical. The management issues include alignment with customer priorities, maintenance resource planning and allocation, skill training of maintenance personnel, contract maintenance work, and customer satisfaction survey feedback. The technical issues include incident reporting, technical problem resolution, impact analysis, standardization of application procedures and testing practices, software maintainability assessment and test efficiency measurements.

Software enhancement is part of the software evolution process. Software system in field operation is noted for its increasing complexity due to modification and enhancement work done to meet customer needs, continuous changes in maintenance support strategies due to competitive service offerings, and the need to develop the skills and techniques to accommodate the changing business environments. The extent and achievement of software maintenance and enhancement effort should be verified, validated and documented. The specific resources needed for software maintenance should be part of the dependability assurance strategy.

6.4.7 Software documentation

Software documentation is the written text and information of the design and application documents associated with the software product. Documentation is an important part of software engineering. The major categories of software documentation include the following.

a) Architecture and design documentation

The documentation presents an overview of the software product and its relations to application environment and construction principles to be used in the design. The software design document is a comprehensive software design model providing detailed information.

- The *data design* describes the structures of the software. Attributes and relationships between data objects dictate the choice of data structures, which impact the structural complexity of the software affecting dependability in performance operation.
- The *architectural design* uses information flow characteristics, and maps them into the program structure, which impacts the modularity of the software design affecting reliability of software module design. Recommended practice for architectural description [32] should be implemented.
- The *interface design* describes the internal and external program interfaces as well as the design of human interfaces including hardware interfaces and hardware drivers. Internal and external interface designs are based on the information obtained from the design analysis, which affects redundancy schemes and reliability requirements of systems, software, and hardware module designs and configurations.
- The *procedural design* describes the structured programming concepts using graphical, tabular, and textual notations. These design media enable the designer to represent procedural details that facilitate translation to code which affects consistency in programming practices and reduction in coding errors introduction.

b) Technical documentation

Technical documentation includes code, algorithms, interfaces and additional text to describe various aspects of the software products intended operation. The documentation should be comprehensive but concise in writing the source code to facilitate software maintenance and update. In-code commenting is a form of technical documentation, where the commenting includes brief explanatory comments lines added to the code that are recognized by the compiler to be comments only and not part of the code for execution. The purpose of in-code commenting as a software documentation practice is to increase reusability and maintainability. This would facilitate code review and inspection, code update and modification, and enhance software integrity and reliability. Technical documentation can also include where needed and applicable to the project, such as requirements specifications, test plans and procedures, technical reports and relevant data.

c) User documentation

User documentation [33] includes manuals for the end users, system administrators and support personnel. It is aimed at assisting the end user application of the software products. The documentation describes how the software can be used in its application environment. User documentation also describes the features of the software product, and assists the user in realizing these features for application; including software release and version control information, trouble-shooting guidelines, safety instructions, warnings and restrictions on the use of the software, and help instructions. Sometimes on-line help is available to promote user-friendly access and service contact to achieve customer satisfaction.

d) Marketing documentation

Marketing documentation includes promotional materials to encourage casual observers to learn more about the software product. Web access and customer care centres are common service provisions in today's competitive market environments for obtaining software products and application information. Customer focus is essential in developing a marketing strategy. Marketing documentation is but one of the many approaches for dissemination of information. Marketing documentation can include information addressing specific dependability values and related issues for appropriate software applications such as software reuse and modification for specific applications.

6.4.8 Automated tools

Automated tools are useful for routine data processing, computational analysis, and comparison of evaluation results. There are broad ranges of automated tools available in the market to meet most application needs for modelling, analysis, and knowledge-base data to support all forms of dependability assessment. The selection and application of appropriate tools for specific project tasks would require the knowledge and experience of the dependability engineer or practitioner. Automated tools are enabling systems that could help routine computational work to improve productivity. The validity and accuracy of these automated tools should be investigated prior to commitment for project application. The supportability of these tools should be determined before tool acquisition. Automated tools are used for software development and testing applicable to dependability enhancement and reliability growth improvement. They form an essential part of the enabling system for application in software verification and software system validation.

6.4.9 Technical support and user training

Technical support is a range of services providing assistance with the software products in use. The objective is to help the user solve specific problems with product operation or application. Technical support takes various forms including telephone query, online service, e-mail, remote access repair and on-site visit for problem solving. There are increasing growth and use of outsourced call centres by technology product development organizations for business, economic and geographical reasons to facilitate real-time response to technical support services. These call centres serve as centralized technical support for a broad range of technology products such as computer systems including software requiring technical assistance around the clock with worldwide toll-free user access. Technical support services form part of the maintenance support to sustain product operability and reliability performance contributing to dependability improvement.

Software user training is an important aspect of software dependability improvement. The objective is to enhance or familiarize the skill level or understanding of the software product applications from the users' perspective. Software user training takes various forms including online access of the product supplier's tutorial database, call centre assistance, dedicated technical expert service to address unusual problems encountered.

7 Software assurance

7.1 Overview of software assurance

Software assurance is the planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures. The capability maturity models [8, 9] are common management tool recommended for implementation of software assurance programs in software development organizations. There are also extensive documented software assurance methodology and procedures for software development and applications [34].

Software assurance generally involves the technical disciplines of quality, reliability, safety and security associated with software product development and system operation. The software assurance process is to plan, develop, maintain and provide grounds for confidence and decision making. The assurance life cycle [35] is conducted for conformity assessment purposes throughout the system life cycle on software products to meet applicable safety, security, dependability and other objectives. The assurance case [36] studies are claim records on process performance and the physical properties and functional characteristics of the software system audited for proof of conformance to system specifications. Software assurance engages in risk assessment, verification and validation testing, documentation and maintenance of audit records as objective evidence. Software assurance utilizes relevant project-based measurement data to monitor the software product and relevant process for possible improvements.

Software dependability emphasizes software reliability as an intrinsic part of software assurance through implementation of software reliability engineering process [37]. Software dependability and quality are pre-requisites for achievement of safety and security in system operation.

7.2 Tailoring process

Tailoring is a project management activity to assure timing and action, and appropriate allocation of resources to meet project needs. The tailoring process can be effectively employed for implementation of software assurance activities. Tailoring is often used in short-term projects to enhance or sustain system operation where the project requirements and constraints are more restrictive than starting a new development project. The following tasks are recommended for implementation of the tailoring process.

- Identify and document the circumstances that influence tailoring, such as operating environment, project size and complexity, project schedule and budget, resource availability, safety, security and integrity issues, legacy issues, and standards conformance requirements.
- Identify input requirements for decision-making.
- Establish project objectives and plan the tailoring process for implementation.
- Select appropriate life cycle stages applicable for tailoring to achieve intended results.
- Document tailoring results to facilitate review of effectiveness and improvement.

7.3 Technology influence on software assurance

Software technology has provided numerous advancements for efficient software development resulting in versatility and economic advantages for software applications. Software assurance has traditionally been focusing on software quality and reliability improvements from a product development perspective. Recent cyber attacks in software operations have become more frequent, more prominent and increasingly sophisticated. They affect not only the software developers using the COTS software but also cause significant time-lost problems to software system operators and users of the end products. The entire situation has become a chain reaction propagated by unknown viruses, stealthy intrusions

and cyber attacks creating a complex and dynamic risk environment for IT-based operations that are software dependent.

Software assurance is critical to organizations involved in safety, security and financial transactions in view of the vulnerability in software applications. Software assurance encompasses the development and implementation of methods and processes for ensuring that software functions as intended while mitigating the risks of vulnerabilities, malicious code, faults or errors that could bring harm to the end user. Software assurance is vital to ensuring the security of critical IT resources. With the rapidly changing nature of threat environment, even the highest level of quality software is not impervious from cyber intrusions if the software is improperly configured and maintained. Managing the threats in cyberspace requires a layered approach on security prevention and collaboration. The developers build more secure and robust software, the system integrators ensure that the software is installed correctly, the operators maintain the system properly, and the end users using the software in a safe and secure manner.

This leads to organizations involved with software to redefine software assurance for their operations. For example, *software assurance* can be interpreted as the “level of confidence that software is free from vulnerabilities, either intentionally or unintentionally designed into the software or accidentally inserted at any time during the software life cycle, and that the software functions in the intended manner” [38]. Software assurance should provide a reasonable level of justifiable confidence that the software will function correctly and predictably by a manner consistent with its documented requirements. The assurance objective is to ensure that the software function is not compromised either through direct attack or through sabotage by maliciously implanted code.

Depending on specific applications, the level of confidence in software assurance addresses:

a) *trust-worthiness* – that no exploitable vulnerabilities exist, either maliciously or unintentionally inserted;

b) *predictable execution* – that software functions when executed as intended will provide justifiable confidence;

c) *conformance* – that planned and systematic set of multi-disciplinary activities to ensure software processes and products conforms to requirements, standards and procedures.

The challenges identified for software assurance include:

1) accidental design mistakes or implementation errors that lead to exploitable code vulnerabilities;

2) the changing technological environment which exposes new vulnerabilities and provides the cyber attackers with new tools for exploitation;

3) malicious insiders and outsiders who seek to do harm to the developers or the end users.

The first challenge is accidental and unintentional. The second and third challenges are intentional and deliberate by the cyber attackers. The countermeasure is to manage risks associated with these challenges through software assurance best practices.

7.4 Software assurance best practices

There are software technology and software assurance forums [39] that involve government, industry, academia and user participation in implementation of software assurance best practices. The recommended software development practices are identified in 6.1. The recommended software assurance best practices are presented as follows:

a) establishment of software assurance policy to guide software development and process implementation;

b) training on software product related technology applications and the use of reference resources;

c) use of common software architecture design platform to facilitate diverse software product development;

d) implementation of software life cycle processes;

e) initiation of software assurance case studies for risk assessment where warranted and appropriate;

f) established common criteria for verification and validation for software qualification and conformance;

g) configuration management control of software version release;

h) established software performance and fault tracking and data collection system for software design and process improvement;

i) established customer help centre to facilitate users service support and software product application.

Annex A
(informative)

Categorization of software and software applications

A.1 Categorization of software

A.1.1 Software categories

The categories of software include relevant software development products and data that are produced by the software engineering process. Software characteristics and application environments are influencing factors that affect the dependability processes in software design and implementation. The categorization scheme presents an orderly combination of views and categories related to software [40].

The category is represented by the grouping of software based on its attributes or characteristics. Emphasis is placed on software dependability related issues to facilitate development and applications. Typical examples of such groupings include as follows.

A.1.2 Characteristics

- *Operation mode* – categories defined by specific processing technique or type adopted by the software system such as real-time, batch, time-shared, parallel and concurrent processing. A real time system should focus on response time. Time-shared software should focus on interface specifications.
- *Scale of software* – categories defined by the size (e.g. KSLOC) or complexity (e.g. data flow) of the software and interpreted as small, medium or large; simple or complex. Complex or large software should be decomposed or broken down to smaller sizes to facilitate project control, stepwise testing and integration.
- *Stability* – categories defined by intrinsic evolutionary aspect or stability in terms of software system characteristics such as continually changing, incremental change, or unlikely to change. Continually or incremental changes of software require interface specifications that allow flexibility and stability after each change. Special development models such as spiral model and the waterfall model are often used.
- *Software function* – categories defined by the type of function such as compiler, business transaction processing, word processing, control systems. Business transactions should emphasize security and availability. Control systems should focus on availability, safety and security.
- *Security* – categories defined by the level of unauthorized access protection, audit trail, program and data protection. Emphasis should be on robustness and availability.
- *Reliability* – categories defined by the level of required reliability such as maturity, fault tolerance, and recoverability. Emphasis should be on reliability growth and configuration control for reliability achievement.
- *Performance* – categories defined by the software performance in terms of capacity, throughput, turnaround or response time. Emphasis should focus on response time that varies with load and capacity.
- *Language* – categories defined by the type of programming language primarily used for the software such as traditional (e.g. COBOL, FORTRAN), procedural (e.g. C), functional (e.g. Lisp), object oriented (e.g. C++). Emphasis should focus on programmer training and user familiarity with the programming language features and limitations.

A.1.3 Environment

- *Application area* – categories defined by the type or class of external system in which the software is used such as e-business, process control and networking system. Security and data integrity are major influencing factors to e-business. Safety and reliability are

important concerns in process control. Response time and availability are critical for network systems operation.

- *Computer system* – categories defined by the specific target computer system in which the software operates such as microprocessor controlled, mainframe, and real-time operating system. Limitations of memory size and programming code are important to microprocessor systems. Compatibility of software operability in mainframe hardware configuration and response time for real-time operation should be considered.
- *User class* – categories defined by the skill level or characteristics of its intended user class such as novice, intermediate or expert. User class identification is essential for interface design and user instruction development to facilitate ease of application.
- *Computer resource* – categories defined by the limitations of the computer resource such as memory requirement, disk requirement, and local area network requirement. The limitations of computer resource would affect development of software support needs as well as application capability.
- *Software criticality* – categories defined by the product integrity level requirements such as national security, organizational security, and privacy. Regulatory requirements and societal needs should be taken into consideration.
- *Software product availability* – categories defined by the availability of the software product such as commercial off-the-shelf (COTS), custom or proprietary software. The timing for acquisition and availability of software product is a decision factor for in-house design or outsourcing in project management.

A.1.4 Data

- *Data representation* – categories defined by data item, type and structure such as relational, indexed, formatted file. Data compatibility should be considered.
- *Software data usage* – categories defined by the type of usage of the intended software data such as single user, multiple sequential users. Data usage would affect data file design and data maintenance support criteria.

A.2 Software applications

Software is used in a wide variety of applications. In general, computer software applications can be grouped as follows.

- *System software* provides the infrastructure to control the computer hardware so that application software can perform. Examples are operating systems such as Microsoft Windows, Mac OS and Linux systems.
- *Application software* is the computer software designed to facilitate user in performing a particular task such as word processors, spread sheets and database applications.
- *Firmware* is the software resident in the programmable memory devices of the end-user products for internal control of various electronic devices such as remote controls, calculators, mobile phones, and digital cameras.
- *Middleware* is the computer software that connects software elements for multiple applications or provision of services such as multiprocessing in distributed systems and web services.
- *Testware* is a subset of software with the special purpose for software testing and test automation.
- *Programming software* is software development tool to facilitate software designers to create, debug, maintain, or support other programs and applications. Examples include CASE tools.
- *Malware* is the malicious software designed to infiltrate a computer without the owner's consent.

Annex B
(informative)

Software system requirements and related dependability activities

B.1 General

Typical software system requirements and related dependability activities are summarized for each software life cycle stage. The information can be used as baseline reference for tailoring of software dependability projects.

B.2 Requirements definition

Software system requirements	*Related dependability activities*
• Market information on software products • System application requirements and user needs • Operating system domain and platform	• Identify software requirements • Identify performance needs • Identify support needs

B.3 Requirements analysis

Software system requirements	*Related dependability activities*
• Functional and capability performance requirements • Application scenarios • Application specific requirements for safety, security and integrity where applicable • Interface requirements • Qualification requirements • Feasibility of software design and testability • Feasibility of operation and maintenance • Installation and acceptance requirements • Documentation requirements	• Develop operational profile • Develop dependability project plan • Develop dependability assurance plan • Identify software dependability metrics • Determine data integrity requirements • Determine safety and security requirements • Establish human-factors engineering (ergonomics) design rules • Establish software support criteria • Identify constraints affecting dependability design and implementation including application specific requirements for design incorporation • Establish software reuse criteria • Establish software reliability growth and qualification acceptance criteria • Determine reliability test records and documentation requirements

B.4 Architectural design

Software system requirements	*Related dependability activities*
• An architecture describing the top-level structure and identifying the constituent software elements • Requirements transformation and allocation to facilitate configuration of the software items • Incorporation of application specific requirements for safety, security and integrity where needed in the system architecture • Internal and external interfaces for system integration and verification • Preliminary documentation for database and test requirements • Recommended design methods and standards to meet project objectives and design specifications • Traceability to the requirements of the software item • Feasibility of detailed design • Operation and maintenance conditions	• Perform application scenario analysis • Determine software structural and functional complexity • Incorporate application specific requirements in modelling system dependability performance • Perform availability/reliability functional model analysis • Perform software availability/reliability allocation • Establish dependability metrics database • Conduct preliminary availability/reliability prediction • Establish software reliability growth and qualification acceptance plan • Establish data records and reporting system • Establish software support plan • Review architectural design for implementation

B.5 Detailed design

Software system requirements	*Related dependability activities*
• A refined lower-level structure for coding of software units for inclusion in software configuration items • Detailed design specifications of software units and descriptions of software configuration items • Consistency and traceability of detailed design and architectural design specifications • Establishment of design methods and standards to meet project requirements • Establishment of special design methods to address safety, security and integrity issues where applicable • All interface requirements for compilation and testing of software units and configuration items • Documentation of database and detailed test requirements and test schedules • Project management reviews to monitor progress and delivery targets • Baseline for software configuration, and communications of design changes	• Implement software design rules • Establish measurement standards and metric evaluation criteria • Incorporate specific designs to meet safety, security and integrity requirements • Foster fault tolerant design • Apply software dependability standards • Conduct software code review and inspection • Refine software availability/reliability allocation • Perform software complexity assessment • Predict software unit reliability • Predict software subsystem availability/reliability • Perform design trade-off analysis • Refine software availability/reliability prediction • Update dependability metrics database • Implement configuration management • Conduct formal design review • Conduct project review

B.6 Realization

Software system requirements	*Related dependability activities*
• Software unit design and coding methods and standards • Software configuration item with specific software units • Verification criteria for unit test • Test coverage of software units • Verification of software functions including application specifications for safety, security and integrity requirements • Feasibility of software integration and testing	• Implement measurement standards and metric evaluation criteria • Determine code coverage of software units • Perform unit testing • Determine fault coverage and test completeness • Categorize fault data for classification • Implement dependability assurance process for unit and functional testing • Verify software units and functions in meeting performance and application specifications • Establish failure reporting, analysis and corrective action system • Implement software assurance program including outsourcing and supply chain where needed • Conduct project review

B.7 Integration

Software system requirements	*Related dependability activities*
• Integration strategy for software units and configuration • Verification criteria for software configuration item test • Verification of software subsystems including application specifications for safety, security and integrity requirements • Documentation of integration test results • Documentation of design changes • Regression strategy for re-verification of changed items • Test data collection system	• Implement fault tracking procedure • Implement fault analysis procedure • Initiate reliability growth program • Implement failure reporting, analysis and corrective action system • Implement data collection system • Verify software subsystems for integration • Perform integration testing • Evaluate availability/reliability test data • Identify problem areas • Perform corrective actions • Control design change and version release • Conduct project review

B.8 Acceptance

Software system requirements	*Related dependability activities*
• Criteria for software system acceptance • Demonstration of test compliance • Validation of integration test results met requirements • Validation of software system for customer acceptance • Regression strategy for re-testing of integrated software changes • Documentation of qualification acceptance results	• Perform reliability growth testing and accelerated testing as required • Monitor reliability trend and improvement status • Perform qualification testing • Review test results for acceptance • Initiate customer acceptance • Validate software system in meeting customer requirements including dependability performance demonstration and safety, security and integrity performance features where applicable • Document software version release status

B.9 Operation and maintenance

Software system requirements	*Related dependability activities*
• Operation procedures and conditions • Maintenance support strategy • Logistic support • Field data collection • User training • Software assurance program to sustain dependability of system operation	• Monitor field performance trends • Update field performance and maintenance support records • Conduct customer satisfaction surveys • Review field data to identify areas for reliability improvement • Establish field performance operational profile • Maintain system dependability performance history and experience database • Collect appropriate dependability metrics for reliability forecasting • Implement software assurance best practices

B.10 Software update/enhancement

Software system requirements	*Related dependability activities*
• Software upgrades • Perfective maintenance strategy implementation • New service introduction and impact assessment • Effects of enhancement/improvement on software performance	• Monitor software upgrades • Conduct perfective maintenance • Implement design change and configuration control • Assess new service introduction impact • Manage new software version release

B.11 Retirement

Software system requirements	*Related dependability activities*
• Termination of specific service • User advisory of termination of old service and new service replacement	• Identify retired software and support service termination • Advise customer care service of any required dependability actions

Annex C
(informative)

Capability maturity model integration process

Capability maturity model integration (CMMI) is a process improvement maturity model for the development of products and services. It consists of best practices that address development and maintenance activities covering the product life cycle from conception through delivery and maintenance. The *CMMI for development* [9] models contain practices that cover project management, process management, systems engineering, hardware engineering, software engineering, and other supporting processes used in development and maintenance. The CMMI process correlates with the implementation of software system requirements and related dependability activities as shown in Annex B. CMMI is used for benchmarking and appraisal activities, as well as guiding an organization's improvement efforts. The CMMI process is designated by levels.

- Capability levels, which belong to a continuous representation, apply to an organization's process improvement achievement in individual process areas. These levels are means to guide incremental improvement process corresponding to a given process area. There are six capability levels, numbered from 0 through 5.
- Maturity levels, which belong to a staged representation, apply to an organization's process improvement achievement across multiple process areas. These levels are used for predicting the general outcomes of the next project undertaken. There are five maturity levels, numbered from 1 through 5.

Table C.1 aligns the six capability levels to the five maturity levels for comparison.

Table C.1 – Comparison of capability and maturity levels

Level	Continuous representation capability levels	Staged representation maturity levels
0	An *incomplete process* is a process that is either not performed or partially performed. One or more of the specific goals of the process area are not satisfied, and no generic goals exist for this level since there is no reason to institutionalize a partially performed process.	N/A
1	A *performed process* is a process that satisfies the specific goals of the process area. It supports and enables the work needed to produce work products. Although capability level 1 results in important improvements, those improvements can be lost over time if they are not institutionalized. The application of institutionalization helps to ensure that improvements are maintained.	At maturity level 1, processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment to support the processes. Success in these organizations depends on the competence of the people in the organization and not on the use of proven processes. The outcomes of maturity level 1 organizations often produce products and services that work; however, they frequently exceed their budgets and do not meet their schedules. There is a tendency to over commit, abandonment of processes in a time of crisis, and an inability to repeat their successes.

Level	Continuous representation capability levels	Staged representation maturity levels
2	A *managed process* is a performed process that has the basic infrastructure in place to support the process. It is planned and executed in accordance with the policy; employs skilled people who have adequate resources to produce controlled outputs; involves relevant stakeholders; is monitored, controlled, and reviewed; and is evaluated for adherence to its process description. The process discipline reflected by capability level 2 helps to ensure that existing practices are retained during times of stress.	At maturity level 2, the projects of the organization have ensured that processes are planned and executed in accordance with the policy; the projects employ skilled people who have adequate resources to produce controlled outputs; involve relevant stakeholders; are monitored, controlled, and reviewed; and are evaluated for adherence to their process descriptions. The outcomes of maturity level 2 organizations ensure that existing practices are retained during times of stress; projects are performed and managed according to their documented plans; the status of the work products and the delivery of services are visible to management at defined points at major milestones and at the completion of major tasks. Commitments are established among relevant stakeholders and are revised as needed. Work products are appropriately controlled. The work products and services satisfy their specified process descriptions, standards, and procedure.
3	A *defined process* is a managed process that is tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process improvement information to the organizational process assets. The relevant standards, process descriptions, and procedures are consistent and tailored to suit a particular project or organizational unit. A defined process clearly states the purpose, inputs, entry criteria, activities, roles, measures, verification steps, outputs, and exit criteria. The processes are managed proactively by understanding the interrelationships of the process activities and detailed measures of the process, its work products, and its services.	At maturity level 3, the processes of the organization are well characterized and understood, and are described in standards, procedures, tools, and methods. These standard processes are used to establish their consistency in implementation across the organization. Projects establish their defined processes by tailoring the organization's set of standard processes according to tailoring guidelines. The outcomes of maturity level 3 organizations demonstrate consistencies in performance.
4	A *quantitatively managed process* is a defined process that is controlled using statistical and other quantitative techniques. Quantitative objectives for quality and process performance are established and used as criteria in managing the process. Quality and process performance is understood in statistical terms and is managed throughout the life of the process.	At maturity level 4, the organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance is understood in statistical terms and is managed throughout the life of the processes. For selected sub-processes, detailed measures of process performance are collected and statistically analysed. Quality and process performance measures are incorporated into the organization's measurement repository to support fact-based decision-making. Special causes of process variation are identified and, where appropriate, the sources of special causes are corrected to prevent future occurrences. The outcomes of maturity level 4 organizations demonstrate adequate control of performance of processes by using statistical and other quantitative techniques to ensure performance results are quantitatively predictable.

Level	Continuous representation capability levels	Staged representation maturity levels
5	An *optimizing process* is a quantitatively managed process that is improved based on an understanding of the common causes of variation inherent in the process. The focus of an optimizing process is on continually improving the range of process performance through both incremental and innovative improvements.	At maturity level 5, the organization continually improves its processes based on a quantitative understanding of the common causes of variation inherent in processes; focuses on continually improving process performance through incremental and innovative process and technological improvements. Quantitative process improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities. The outcomes of maturity level 5 organizations demonstrate continual process improvement to achieve the established quantitative process improvement objectives.

Annex D
(informative)

Classification of software defect attributes

D.1 General

The *orthogonal defect classification* (ODC) is a method used for capturing and grouping of software fault information in terms of software defect attributes. The ODC process provides the capability to extract defect signatures and infer the health of the software development process. The classification is based on what is known about the defect. When a fault or defect is opened, the way in which the defect was found and exposed and the impact to the user are normally known. Therefore, the ODC attributes of *Activity*, *Trigger*, and *Impact* can be classified. Similarly, when a fault is diagnosed and fixed, the details of the fix are known. The ODC attributes of *Defect Target*, *Defect Type*, *Qualifier*, *Source*, and *Age* can be classified. Additional non-ODC attributes such as the scheduled project phase-found, severity, and component, that are captured in any fault or defect tracking system can be used in conjunction with ODC-based analysis. ODC does not impose a specific structure. The following clauses summarize the classification of software defect attributes under the headings of *opener section*, *closer section* and *activity to trigger mapping*.

D.2 Opener section

An opener section is to classify attributes when a fault is found.

When a fault is discovered and is open for diagnosis during software development, the information on fault exposure and likely impact and severity can be assessed. Typical attributes of fault information captured when a fault is found can be summarized in Table D.1.

Table D.1 – Classification of software defect attributes when a fault is found

Fault removal activity [a] **(when detected)**	**Trigger** [b] **(how detected)**	**Impact** [c] **(effect and severity)**
• Design review • Code inspection • Unit test • Function test • System test • Acceptance test • Qualification test • Reliability growth test	• Design conformance • Compatibility • Concurrency • Coverage • Sequencing • Interaction • Configuration	• Installability • Serviceability • Integrity/Security/safety • Reliability • Maintenance • Accessibility • Usability
NOTE 1 Software faults are often hard to replicate. It is important to capture the circumstances leading up to and surrounding the incidence of the fault detection. NOTE 2 Traceability to requirements is needed; either traceable to the software requirements or traceable to a specific test case.		
[a] *Activity*: actual activity that was being performed at the time the fault was discovered. [b] *Trigger:* the environment or condition that had to exist for the fault exposure. [c] *Impact*: for development faults, the impact is assessed as the potential effect and severity to the user; for field reported fault, the impact is the failure effect and severity to the user.		

D.3 Closer section

A closer section is to classify attributes when a fault is fixed.

When a fault is closed after the fix is applied, the exact nature of the fault and the scope of the fix are known. Typical attributes of fault information captured when a fault is fixed can be summarized in Table D.2.

Table D.2 – Classification of software defect attributes when a fault is fixed

Target [a]	Type [b]	Qualifier [c]	Age [d]	Source [e]
• Design/Code	• Initiation • Checking • Function • Timing • Interface	• Missing • Incorrect • Extraneous	• Base • New • Rewritten	• Developed in-house • Outsourced • Reused from library • Ported

a *Target*: the high level identity of the entity that was fixed.

b *Type*: the nature of the actual correction that was made.

c *Qualifier (applies to the defect Type):* describes the element of either non-existent, or wrong, or irrelevant implementation.

d *Age*: identifies the history of the *Target* such as *Design/Code*, which had the defect.

e *Source*: identifies the origin of the *Target* such as *Design/Code,* which had the defect.

D.4 Activity to trigger mapping

The mappings of activity to trigger group the applicable triggers relevant to the software design review, inspection and test activities. Tables D.3, D.4, D.5, and D.6 show some generic examples of the activity to trigger mappings.

Table D.3 – Design review/code inspection activity to triggers mapping

Activity	Triggers
Design review/code inspection *Reviewing design or comparing the documented design against known requirements*	• *Design conformance* The document reviewer or the code inspector detects the fault while comparing the design element or code segment being inspected with its specification in the preceding stage. This would include design documents, code, development practices and standards, or to ensure design requirements are not missing or ambiguous. • *Logic/flow* The inspector uses knowledge of basic programming practices and standards to examine the flow of logic or data to ensure they are correct and complete. • *Backward compatibility* The inspector uses extensive product/component experience to identify an incompatibility between the function described by the design document or the code, and that of earlier versions of the same product or component. From a field perspective, the customer's application, which ran successfully on the prior release, fails on the current release. • *Lateral compatibility* The inspector with broad-based experience, detects an incompatibility between the function described by the design document or the code, and the other systems, products, services, components, or modules with which it must interface. • *Concurrency* The inspector is considering the serialization necessary for controlling a shared resource when the fault is discovered. This would include the serialization of multiple functions, threads, processes, or kernel contexts as well as obtaining and releasing locks. • *Internal document* There are incorrect information, inconsistency, or incompleteness within internal documentation. Prologues, code comments, and test plans represent some examples of documentation, which would fall under this category. • *Language dependency* The developer detects the defect while checking the language specific details of the implementation of a component or a function. Language standards, compilation concerns, and

Activity	Triggers
	language specific efficiencies are examples of potential areas of concern. • *Side Effects* The inspector uses extensive experience or product knowledge to foresee some system, product, function, or component behaviour which can result from the design or code under review. The side effects would be characterized as a result of common usage or configurations, but outside of the scope of the component or function with which the design or code under review is associated. • *Rare situation* The inspector uses extensive experience or product knowledge to foresee some system behaviour, which is not considered or addressed by the documented design or code under review, and would typically be associated with unusual configurations or usage. Missing or incomplete error recovery would not, in general, be classified with a trigger of *rare situation*, but would most likely fall under *design conformance* if detected during *review/inspection*.

Table D.4 – Unit test activity to triggers mapping

Activity	Triggers
Unit test *White box testing or execution based on detailed knowledge of the code internals*	• *Simple path* The test case was motivated by the knowledge of specific branches in the code and not by the external knowledge of the functionality. This trigger would not typically be selected for field reported defects, unless the customer is very knowledgeable of the code and design internals, and is specifically invoking a specific path (as is sometimes the case when the customer is a business partner or vendor). • *Complex path* In white/grey box testing, the test case that found the defect was executing some contrived combinations of code paths. The tester attempted to invoke execution of several branches under several different conditions. This trigger would only be selected for field reported defects under the same circumstances as those described under *simple path.*

Table D.5 – Function test activity to triggers mapping

Activity	Triggers
Function test *Black box execution based on external specifications of functionality*	• *Coverage* During black box testing, the test case that found the defect was a straightforward attempt to exercise code for a single function, using no parameters or a single set of parameters. • *Variation* During black box testing, the test case that found the defect was a straightforward attempt to exercise code for a single function but using a variety of inputs and parameters. These might include invalid parameters, extreme values, boundary conditions, and combinations of parameters. • *Sequencing* During black box testing, the test case that found the defect executed multiple functions in a very specific sequence. This trigger is only chosen when each function executes successfully when run independently, but fails in this specific sequence. It is also possible to execute a different sequence successfully. • *Interaction* During black box testing, the test case that found the defect initiated an interaction among two or more bodies of code. This trigger is only chosen when each function executes successfully when run independently, but fails in this specific combination. The interaction involves more than a simple serial sequence of the executions.

Table D.6 – System test activity to triggers mapping

Activity	Triggers
System test *Testing or execution of the complete system, in the real environment, requiring all resources*	• *Workload/stress* The system is operating at or near some resource limit, either upper or lower. These resource limits can be created by means of a variety of mechanisms, including running small or large loads, running a few or many products at a time, letting the system run for an extended period of time. • *Recovery/exception* The system is being tested with the intent of invoking an exception handler or some type of recovery code. The defect would not have surfaced if some earlier exception had not caused exception or recovery processing to be invoked. From a field perspective, this trigger would be selected if the defect were in the system or product ability to recover from a failure, not the failure itself. • *Start-up/restart* The system or subsystem was being initialized or restarted following some earlier shutdown or complete system or subsystem failure. • *Hardware configuration* The system is being tested to ensure functions execute correctly under specific hardware configurations. • *Software configuration* The system is being tested to ensure functions execute correctly under specific software configurations. • *Blocked test (normal mode)* The product is operating well within resource limits and the defect surfaced while attempting to execute a system test scenario. This trigger would be used when the scenarios could not be run because there are basic problems, which prevent their execution. This trigger must not be used in customer reported defects.

Annex E
(informative)

Examples of software data metrics obtained from data collection

E.1 Fault data metrics

Metric	Application
Problem reporting data • Date and time fault detected • Detected fault description • Fault detected in program area • Person detected the fault • Fault symptom and status • Severity and priority	Data collected on software projects should be used for reporting problem on fault identification and occurrence
Corrective action data • Date fault corrected • Person corrected the fault • Maintenance action taken • Description of modification • Identification of modules modified • Version control information • Time required to correct fault • Date verified as fault corrected • Person verified the correction	Data collected on corrective actions and verified as fault corrected should be used for reporting problem resolution.
Cumulated faults detected	Cumulated faults detected data should be used to determine fault rate and reliability trend over a period of time
Cumulated faults corrected	Cumulated faults corrected data should be used to determine known faults that require corrective actions and tracking the effectiveness of maintenance actions
Faults detection rate	Faults detection rate is used to indicate trend to facilitate planning of maintenance strategy and resource management
Faults correction rate	Faults correction rate is used to indicate trend to facilitate planning of maintenance strategy and resource management. Priority setting for maintenance action is based on the severity of the fault problem
Faults per location	Fault tracking according to software functions to identify specific area of the code is more error-prone
Criticality of faults	Classifying the degree of impact of faults to set priority for maintenance actions
Number and percentage of severe faults	Indications for planning maintenance strategy
Structural complexity per location	Use with other metrics to determine impact of faults generated related to the complexity of the software structure and location
Functional complexity per location	Use with other metrics to determine impact of faults generated related to the complexity of the software functions and location

E.2 Product data metrics

Metric	Application
Number and percentage of modules that perform more than one function	Indication of cohesiveness of the overall software design on functional complexity. High complexity module will result in low cohesiveness that would require redesign
Number and percentage of modules that have a high structural complexity	Indication of the overall software design requiring redesign to reduce complexity
Number and percentage of modules that have exactly one entrance and one exit	Indication of a cohesive design that should be used as basis for structured design practice
Number and percentage of modules that are documented according to standards	Indication of code completeness that should be used to determine if the code contains all the requirements and addresses the requirements completely
Number and percentage of faults that are found in reused code	Indication of the unreliability of the reused code

E.3 Process data metrics

Metric	Application
Faults introduced by life cycle stage	Indication of when and at what stage the faults were introduced, and to take appropriate actions.
Faults detected by life cycle stage	Indication of when and at what stage the faults were detected, and justification for delay corrective actions for fault removal.
Total time spent in analysis	Indication of the time spent on analysis for problem identification and isolation for corrective action, and the associated resources required.
Total time spent in design	Indication of the time spent for software design, and the associated resources required.
Total time spent in coding	Indication of the time spent for coding and programming, and the associated resources required.
Total time spent in unit testing	Indication of the time spent on unit testing, and the associated resources required.
Total time spent in system testing	Indication of the time spent on system testing, and the associated resources required.
Total maintenance time	Indication of the time spent on maintenance activities, and the associated resources required.
Average maintenance administration time	Indication of the time spent on maintenance administration, and the associated resources required. Maintenance administrative duties include before and after the fault is corrected, such as time spent in assigning maintenance personnel, release of correction in a new version.
Average corrective action time	Indication of the time spent on corrective actions, and the associated resources required. This reflects the cost-effectiveness in the maintenance activities.
Reason for corrective action	This is used to determine the source of faults. Typical reasons include: • previous maintenance action • new requirement • requirement change • misinterpreted requirement • missing requirement • ambiguous requirement • change in software environment • change in hardware environment • code/logic error • performance error
Cost of corrective action	Indication of the total cost of corrective action including fault isolation, problem resolution, and administration for effective maintenance action.

Metric	Application
Percentage of functions tested and verified	Indication of test coverage, test efficiency and completeness.
Percentage of independent paths tested and verified	Indication of test coverage of structural testing and completeness.
Percentage of source lines of code test and verified	Indication of test coverage of software code and completeness.
Historical data	Provision of data history on problem areas related to design, process and product issues.

Annex F
(informative)

Example of combined hardware/software reliability functions

A monitoring control system is shown as a combined hardware/software system consisting of the electronic operation and application functions represented by the system hierarchy. The following provides the basic system hardware and software functional descriptions:

- hardware actuation function to operate the monitoring control mechanism;
- actuation electronic circuit components;
- hardware sensing function to start or stop the activation;
- sensing electronic circuit components;
- software application functions for controlling the rapid initiation and release of the hardware sensing function including
  - software unit for controlling initiation of activation;
  - software unit for controlling release and deactivation;
  - software unit for monitoring the speed of activation and deactivation.

These functions are represented by block diagram as shown in Figure F.1.

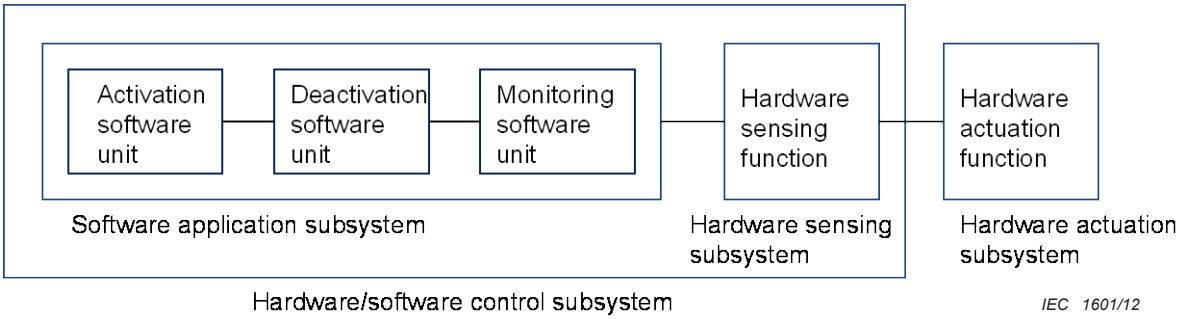

Figure F.1 – Block diagram for a monitoring control system

The *hardware actuation function* can be considered as a hardware subsystem with the complete actuation electronic circuitry comprising of electronic components.

The *hardware sensing function* can be considered as a hardware subsystem with the complete sensing electronic circuitry comprising of electronic components.

The *software application functions* consist of three separate software units designed to work together with the host hardware sensing subsystem. Each of the software unit is designed to perform exactly one function as designated. In order for the hardware sensing subsystem to work, it is necessary that all the software units are incorporated into the software application subsystem to perform as a combined hardware/software subsystem.

For reliability analysis the following should be noted in determining system and subsystem failure rates. Assumption is made of constant failure rates of electronic components, which comprise all hardware functions.

- The hardware actuation function failure rate can be determined by the sum of the failure rates of individual components of the actuation circuitry assuming no redundancy in design and operating continuously full time (λ_1).

- The hardware sensing function failure rate can be determined by the sum of the failure rates of individual components of the sensing circuitry assuming no redundancy in design and operating continuously full time (λ_2).
- The reliability of the software application functions is determined by the combined reliability of the three software units due to their dependency associated with the monitoring control system operational profile. The operational profile does not require full calendar time for execution of these software units, but only needed for their part-time applications upon demand. The execution time associated with each of the software units should be incorporated in the calculation of fault density relevant to each respective software unit. The fault density is then converted to calendar time to determine the effective failure rate. Fault density is used for repairable items and failure rate is used for non-repairable items. Since software is non-repairable, failure rate applies here. In this respect, the entire software application functions are treated as one single software configuration item with failure rate (λ_3).
- The reliability of the monitoring control system in terms of failure rate is determined by the sum of $\lambda_1 + \lambda_2 + \lambda_3 = \lambda$.

Annex G
(informative)

Summary of software reliability model metrics

There are many software reliability models existing today and used in industry practice. Most models have been developed for specific applications. Most of them have been computerized and automated to facilitate data processing, analysis and evaluation. There is no one single model that would fit all applications. It is up to the user to select and apply appropriate software reliability models to meet their own project needs. The following presents a list of common metrics used in most software reliability models.

a) Total number of inherent faults in the software. This metric is assumed to be fixed and finite.

b) Total number of latent faults (bugs) in the software. This metric is assumed to be variable because of the possibility of inserting new faults into the code over time.

c) Total number of faults corrected at some point in time, or after some usage or testing time has elapsed.

d) Total number of faults detected at some point in time, or after some usage or testing time has elapsed.

e) Number of testing periods or intervals. This is the number of intervals between fault correction activities. Some models assume that faults are corrected as soon as they are detected.

f) Total number of faults corrected up to a certain testing period.

g) Change in the failure rate.

h) Testing or usage time accumulated up to the present time or present number of detected faults.

i) Execution time accumulated.

j) Initial failure rate.

k) Present failure rate.

l) Growth rate.

m) Estimated number of lines of executable code at testing or usage time.

n) Total number of test cases run.

o) Total number of successful test cases run.

Annex H
(informative)

Software reliability models selection and application

There are many models and metrics available today for estimating software reliability and measuring characteristics of software. All reliability models are developed for curve fitting exercises using the metric data collected for the model inputs. The validity and accuracy of the model application and resultant output depend on the assumptions made in the model formulation and the relevancy of the data input to the model to generate the output. Most models are developed to meet a specific need during the software life cycle. Examples include prediction model during software design, and estimation model to determine additional test time required before software release. Some models are developed to predict the reliability of software before the code is written. Data input for reliability prediction in such case is often based on historic data of similar software system and application. Other models are deployed for estimation of software reliability growth trends based on interim test data input. There is no one single model that is capable of covering the entire spectrum of the software life cycle. In practice, several models are often tried and used to determine software reliability. Statistical techniques, such as goodness of fit test to check how well a model fits a set of observations, are often used for model selection. Most software reliability models are automated due to their iterative computational needs. Interpretations of reliability modelling results require practical experience and reliability modelling expertise.

Table H.1 presents some examples of software reliability models used in industry practice. It is not the intention of this standard to provide detailed model formulation and parametric applications. References on software reliability models and their specific applications are well documented in the literature [14, 37].

Table H.1 – Examples of software reliability models

	Model name	Assumptions	Data requirements	Application limitations
1	Musa basic	• Finite number of inherent errors (latent faults) • Constant error rate over time •Exponential distribution	• Number of detected faults at some point in time • Estimate of initial failure rate • Software system present failure rate	• Software is operational • Use after system integration • Assume no new faults are introduced in correction • Assume number of residual faults decreases linearly over time
2	Musa-Okumoto	• Infinite number of inherent errors (latent faults) • Changing error rate over time •Logarithmic distribution	• Number of detected faults at some point in time • Estimate of initial failure rate • Relative change of failure rate over time • Software system present failure rate	• Software is operational • Use for unit to system tests • Assume no new faults are introduced in correction • Assume number of residual faults decreases exponentially over time
3	Jelinski-Moranda	• Finite and constant number of inherent errors (latent faults) • Constant error rate over time • Errors corrected as soon as detected • Binomial exponential distribution	• Number of corrected faults at some point in time • Estimate of initial failure rate • Software system present failure rate	• Software is operational • Use after system integration • Assume no new faults are introduced in correction • Assume number of residual faults decreases linearly over time
4	Littlewood-Verrall	• Uncertainty in correction process	• Estimate of the number of failures • Estimate of the reliability growth rate • Time between failures detected or the time of the failure occurrence	• Software is operational
5	Schneidewind	• No new faults are introduced in correction	• Estimate of failure rate at start of first interval • Estimate of proportionality constant of failure rate over time • Faults detected in equal time interval	• Software is operational • Rate of fault detection decreases exponentially over time
6	Geometric	• Inherent number of faults to be infinite	• Decreasing geometric progression function as failures are detected • Time between failure occurrences or time of failure occurrence	• Software is operational • Faults are independent and unequal in probability of occurrence and severity
7	Brooks-Motley	•Rate of fault detection constant over time	• Test effort of each test • Probability of fault detection in i^{th} test • Probability of correcting faults without introducing new ones • Number of faults remaining at start of i^{th} test • Total number of faults found in each test	• Software developed incrementally • Some software modules have different test effort than others

	Model name	Assumptions	Data requirements	Application limitations
8	Bayesian	• Software is relatively fault free	• Length of testing time for each interval • Number of failures detected in each interval	• Software is operational • Software is corrected at end of testing interval
9	Keene	• Correlates the delivered latent fault content with the development process capability and software size (KSLOCs)	• CMM maturity level to assess process capability, estimated KSLOCs of deliverable code, estimated number of months to reach maturity after release, fault latency, per cent of severity 1 and 2 faults, recovery time, use hours per week of the code, and per cent fault activation	• Per cent fault activation is an estimated parameter that represents the average percentage of seats of system users that are likely to experience a particular fault; the CMMI capability level should be assessed for the development organization as well as the maintenance organization

The following criteria should be used to facilitate model selection:

- failure profiles;
- maturity of software product;
- characteristics of software development;
- characteristics of software test;
- existing metrics and data.

For model execution, the use of automated computational tools is recommended. There are commercially available tools that cover some or all of the software reliability models identified in Table H.1. The main advantage of using automated computational tools is the time and cost savings of implementation for model applications. By choosing an appropriate tool it is possible to compare the results of a set of data runs on several models to determine best fit.

The following criteria should be considered in selecting a tool or tools for an organization:

- availability of the tool compatible with the organization’s computer systems;
- cost of installing and maintaining the program;
- number of studies likely to be carried out for tool applications;
- types of software systems to be studied;
- quality of the tool documentation;
- ease of learning the tool;
- flexibility and power of the tool;
- technical support of the tool.

Bibliography

[1] IEC 62508, *Guidance on human aspects of dependability*

[2] ISO/IEC 12207, *Systems and software engineering – Software life cycle processes*

[3] IEC 60300-1, *Dependability management – Part 1: Dependability management systems*

[4] IEC 60300-2, *Dependability management – Part 2: Guidelines for dependability management*

[5] KLINE, M.B., *Software and hardware R&M: What are the differences?* Proceedings of the Annual Reliability and Maintainability Symposium, 1980

[6] LIPOW, M., SHOOMAN, M. L., *Software reliability, Tutorial Session*, Annual Reliability and Maintainability Symposium, 1986

[7] ISO/IEC 15288, *Systems and software engineering – System life cycle processes*

[8] *Capability Maturity Model® (CMM®)*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA USA

[9] *Capability Maturity Model Integration® (CMMI®) for Development, Version 1.2*; Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA USA 2006

[10] IEC 60300-3-3, *Dependability management – Part 3-3: Application guide – Life cycle costing*

[11] IEC 62347, *Guidance on system dependability specifications*

[12] IEC 61160, *Design review*

[13] CHILLAREGE, Ram, *Orthogonal Defect Classification – A concept for in process Measurements*, IEEE Transactions on Software Engineering, 1992

[14] LYU, M.R. (Ed.): *The Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill Book Company, 1996

[15] IEEE-1633: *Recommended Practice on Software Reliability, 2009*

[16] ISO/IEC 20926, *Software and systems engineering — Software measurement — IFPUG functional size measurement method 2009*

[17] IEC 61078, *Analysis techniques for dependability – Reliability block diagram and boolean methods*

[18] IEC 61025, *Fault tree analysis (FTA)*

[19] IEC 61165, *Application of Markov techniques*

[20] IEC 62551, *Analysis techniques for dependability – Petri net techniques*[2]

[21] DUGAN, J.B., *Fault Tree Analysis for Computer-based Systems, Tutorial Session*, Annual Reliability and Maintainability Symposium, 2000

[22] IEC 62198, *Project risk management – Application guidelines*

[23] IEC 60812, *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*

[24] IEC 60300-3-1, *Dependability management – Part 3-1: Application guide – Analysis techniques for dependability – Guide on methodology*

[25] ISO/IEC 15026-3, *Systems and software engineering – System and software assurance – Part 3: System integrity levels*

[26] IEC 61508-3, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*

[27] ISO/IEC 13335-1, *Information technology – Security techniques – Management of information and communications technology security – Part 1: Concepts and models for information and communications technology security management*
(withdrawn)

[28] IEC 62429, *Reliability growth – Stress testing for early failures in unique complex systems*

[29] IEC 61014, *Programmes for reliability growth*

[30] IEC 61164, *Reliability growth – Statistical test and estimation methods*

[31] IEC 62506, *Methods for product accelerated testing*[2]

[32] ISO/IEC/IEEE 42010, *Systems and software engineering – Architectural description*

[33] ISO/IEC 18019, *Systems and software engineering – Guidelines for the design and preparation of user documentation for application software*
(withdrawn)

[34] *Software Assurance Standard*, NASA-STD-8739.8 w/Change 1, May 2005

[35] ISO/IEC 15026-4, *Systems and software engineering – System and software assurance – Part 4: Assurance in the life cycle*[2]

[36] ISO/IEC 15026-2, *Systems and software engineering – System and software assurance – Part 2: Assurance case*

[37] LAKEY, P.B., NEUFELDER, A.M., *System and Software Reliability Assurance Notebook*, Rome Laboratory, 1996

[38] *National Information Assurance (IA) Glossary*, (CNSS Instruction No. 4009), National Security Telecommunications and Information Systems Security Committee (NSTISSC), published by the United States federal government (unclassified), June 2006

2 To be published.

[39] *Software Assurance: An Overview of Current Industry Best Practices*, Software Assurance Forum for Excellence in Code, February 2008

[40] ISO/IEC TR 12182, *Information technology – Categorization of software*

This page deliberately left blank

British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards -based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at bsigroup.com/standards or contacting our Customer Services team or Knowledge Centre.

Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at bsigroup.com/shop, where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to bsigroup.com/subscriptions.

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member**.

PLUS is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit bsigroup.com/shop.

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email bsmusales@bsigroup.com.

BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK

Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

Copyright

Useful Contacts:

Customer Services
Tel: +44 845 086 9001
Email (orders): orders@bsigroup.com
Email (enquiries): cservices@bsigroup.com

Subscriptions
Tel: +44 845 086 9001
Email: subscriptions@bsigroup.com

Knowledge Centre
Tel: +44 20 8996 7004
Email: knowledgecentre@bsigroup.com

Copyright & Licensing
Tel: +44 20 8996 7070
Email: copyright@bsigroup.com

bsi.

...making excellence a habit.™