ICS : 29.280; 45.060.10

geregistreerde belgische norm

NBN EN 50128
1e uitg., 2001

Normklasse : C76

Spoortwegtoepassingen
Communicatie-, signalerings- en processystemen
Programmatuur voor besturings- en beveiligingssystemen

Applications ferroviaires
Systèmes de signalisation, de télécommunication et de traitement
Logiciels pour systèmes de commande et de protection ferroviaire

Bahnanwendungen
Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme
Software für Eisenbahnsteuerungs- und Überwachungssysteme

Railway applications
Communications, signalling and processing systems
Software for railway control and protection systems

Toelating tot publicatie : 20 mei 2001

INLEIDING

Deze Belgische norm is de EN 50128 (03/2001).

Tekst Frans, Engels of Duits in 3 bundels (indien beschikbaar)

Commissie : SC 9XA - COMMUNICATION,SIGNALLING AND PROCESSING SYSTEMS (=IEC COM 9)

Goedgekeurd door het BEC op 1 november 2000

Belgisch Elektrotechnisch Comité (BEC) - vereniging zonder winstoogmerk
DIAMANT BUILDING - A. Reyerslaan, 80 - 1030 BRUSSEL
E-Mail: centraloffice@bec-ceb.be - Tel. 02 706 85 70 - prk 000-0105058-07

Belgisch Instituut voor Normalisatie (BIN) - vereniging zonder winstbejag
Brabançonnelaan, 29 - 1000 BRUSSEL Tel. 02 738 01 11 - prk 000-0063310-66

Prijsgroep : 35

EUROPEAN STANDARD **EN 50128**

NORME EUROPÉENNE

EUROPÄISCHE NORM March 2001

ICS 29.280; 45.060.10

English version

Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems

Applications ferroviaires -
Systèmes de signalisation, de
télécommunication et de traitement -
Logiciels pour systèmes de commande
et de protection ferroviaire

Bahnanwendungen -
Telekommunikationstechnik, Signal-
technik und Datenverarbeitungssysteme -
Software für Eisenbahnsteuerungs- und
Überwachungssysteme

CENELEC

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

Central Secretariat: rue de Stassart 35, B - 1050 Brussels

Foreword

This European Standard was prepared by SC 9XA, Communication, signalling and processing systems, of Technical Committee CENELEC TC 9X, Electrical and electronic applications for railways.

The text of the draft was submitted to the formal vote and was approved by CENELEC as EN 50128 on 2000-11-01.

The following dates were fixed:

- latest date by which the EN has to be implemented at national level by publication of an identical national standard or by endorsement	(dop)	2001-11-01
- latest date by which the national standards conflicting with the EN have to be withdrawn	(dow)	2003-11-01

This European Standard should be read in conjunction with EN 50126: "Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)" and EN 50129: "Railway applications - Safety related electronic systems for signalling".

Annexes designated "normative" are part of the body of the standard.
Annexes designated "informative" are given for information only.
In this standard, annex A is normative and annex B is informative.

Contents

Figures

Introduction

This Standard is part of a group of related Standards. The others are EN 50126 "Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)" and EN 50129 "Railway applications - Safety related electronic systems for signalling". EN 50126 addresses system issues on the widest scale, while EN 50129 addresses the approval process for individual systems which may exist within the overall railway control and protection system. This Standard concentrates on the methods which need to be used in order to provide software which meets the demands for safety integrity which are placed upon it by these wider considerations.

This Standard owes much of its direction to earlier work done by Working Group 9 of IEC/TC 65. The work of WG 9 resulted in a generic standard for software for safety systems which is now part of IEC 61508. A particular aspect of the work by WG 9 is its inclusion of Software Integrity Level 0, which covers non-safety software, as well as Software Integrity Levels 1 to 4, which cover safety-related and safety-critical software. This Standard also covers all five Software Integrity Levels.

Account has also been taken of the work of the Institution of Railway Signal Engineers (the IRSE), in particular its Technical Report Number 1, which addressed the same topic.

The key concept of this European Norm is that of levels of software safety integrity. The more dangerous the consequences of a software failure, the higher the software safety integrity level will be.

This European Standard has identified techniques and measures for 5 levels of software safety integrity where 0 is the minimum level and 4 the highest level. Four of these levels, 1 to 4, refer to safety-related software, whilst level 0 refers to non safety-related software. This level has been included as normative in order to allow a smooth transition between software developments for non-safety related systems and those for safety-related systems. The required techniques and measures for each software safety integrity level and for the non safety-related level are shown in the tables. In this version, the required techniques for level 1 are the same as for level 2, and the required techniques for level 3 are the same as for level 4. This European Standard does not give guidance on which level of software integrity is appropriate for a given risk. This decision will depend upon the many factors including the nature of the application, the extent to which other systems carry out safety functions and social and economic factors.

It is the function of EN 50126 and EN 50129 to specify the safety functions allocated to software.

This European Standard specifies those measures necessary to achieve these requirements. The process is illustrated in Figure 1.

EN 50126 and EN 50129 require that a systematic approach be taken to:

i) identifying hazards, risks and risk criteria;

ii) identifying the necessary risk reduction to meet the risk criteria;

iii) defining an overall System Safety Requirements Specification for the safeguards necessary to achieve the required risk reduction;

iv) selecting a suitable system architecture;

v) planning, monitoring and controlling the technical and managerial activities necessary to translate the System Safety Requirements Specification into a Safety-Related System of a validated safety performance (or safety integrity).

As decomposition of the specification into a design comprising safety-related systems and components takes place, further allocation of safety integrity levels is performed. Ultimately this leads to the required software safety integrity levels.

The current state-of-the-art is such that neither the application of quality assurance methods (so-called fault avoiding measures) nor the application of software fault tolerant approaches can guarantee the absolute safety of the system. There is no known way to prove the absence of faults in reasonably complex safety-related software, especially the absence of specification and design faults.

The principles applied in developing high integrity software include, but are not restricted to:

- top-down design methods;
- modularity;
- verification of each phase of the development lifecycle;
- verified modules and module libraries;
- clear documentation;
- auditable documents; and
- validation testing.

These and related principles must be correctly applied. This standard specifies the level of assurance required to demonstrate this at each software safety integrity level.

After the System Safety Requirements Specification, which identifies all safety functions allocated to software and determines the system safety integrity level, has been obtained or produced, the functional steps in the application of this European Standard are shown in Figure 2 and are as follows:

i) define the Software Requirements Specification and in parallel consider the software architecture. the software architecture is where the basic safety strategy is developed for the software and the software safety integrity level (clauses 5, 8 and 9);

ii) design, develop and test the software according to the Software Quality Assurance Plan, software safety integrity level and the software lifecycle (clause 10);

iii) integrate the software on the target hardware (clause 12);

iv) validate the software (clause 13);

v) if software maintenance is required during operational life then re-activate this European Standard as appropriate (clause 16).

A number of activities run across the software development. These include verification (clause 11), assessment (clause 14) and quality assurance (clause 15).

Requirements are given for systems which are configured by application data (clause 17).

Requirements are also given for the competency of staff involved in software development (clause 6).

The standard does not mandate the use of a particular software development lifecycle. However a recommended lifecycle and documentation set are given (clause 7 and Figures 3 and 4).

Tables have been formulated ranking various techniques/measures against the 5 software safety integrity levels. The tables are in annex A. Cross-referenced to the tables is a bibliography giving a brief description of each technique/measure with references to further sources of information. The bibliography is in annex B.

1 Scope

1.1 This European Standard specifies procedures and technical requirements for the development of programmable electronic systems for use in railway control and protection applications. It is aimed at use in any area where there are safety implications. These may range from the very critical, such as safety signalling to the non-critical, such as management information systems. These systems may be implemented using dedicated microprocessors, programmable logic controllers, multiprocessor distributed systems, larger scale central processor systems or other architectures.

1.2 This European Standard is applicable exclusively to software and the interaction between software and the system of which it is part.

1.3 Software safety integrity levels above zero are for use in systems in which the consequences of failure could include loss of life. Economic or environmental considerations, however, may also justify the use of higher software safety integrity levels.

1.4 This European Standard applies to all software used in development and implementation of railway control and protection systems including:

application programming;

operating systems;

support tools;

firmware.

Application programming comprises high level programming, low level programming and special purpose programming (for example: Programmable Logic Controller ladder logic).

1.5 The use of standard, commercially available software and tools is also addressed in this European Standard.

1.6 This European Standard also addresses the requirements for systems configured by application data.

1.7 This European Standard is not intended to address commercial issues. These should be addressed as an essential part of any contractual agreement. All the clauses of this European Standard will need careful consideration in any commercial situation.

1.8 This European Standard is not intended to be retrospective. It therefore applies primarily to new developments and only applies in its entirety to existing systems if these are subjected to major modifications. For minor changes, only clause 16 applies.

2 Normative references

This European Standard incorporates by dated or undated reference, provisions from other publications. These normative references are cited at the appropriate places in the text and the publications are listed hereafter. For dated references, subsequent amendments to or revisions of any of these publications apply to this European Standard only when incorporated in it by amendment or revision. For undated references the latest edition of the publication referred to applies (including amendments).

EN 50126	Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)
EN 50129*	Railway applications - Safety related electronic systems for signalling

* at draft stage

EN 50159-1	Railway applications - Communication, signalling and processing systems Part 1: Safety-related communication in closed transmission systems
EN 50159-2	Railway applications - Communication, signalling and processing systems Part 2: Safety-related communication in open transmission systems
EN ISO 9001	Quality systems - Model for quality assurance in design/development, production, installation and servicing
EN ISO 9000-3	Quality management and quality assurance standards – Part 3: Guidelines for the application of ISO 9001:1994 to the development, supply, installation and maintenance of computer software

3 Definitions

For the purposes of this European Standard, the following definitions apply. For terms not defined here, the following references should be consulted in order of priority:

EN ISO 8402	Quality management and quality assurance – Vocabulary
IEC 60050-191	International Electrotechnical Vocabulary of Chapter 191: Dependability and quality of service
IEEE 610.12	IEEE standard glossary of software engineering terminology
ISO/IEC 2382	Information Technology Vocabulary
ISO/IEC 9126	Information Technology – Software Product Evaluation – Quality characteristics and guidelines for their use

3.1
assessment
process of analysis to determine whether the Design Authority and the Validator have achieved a product that meets the specified requirements and to form a judgement as to whether the product is fit for its intended purpose

3.2
assessor
person or agent appointed to carry out the assessment

3.3
availability
ability of a product to be in a state to perform a required function under given conditions at a given instant of time or over a given time interval, assuming the required external resources are provided

3.4
commercial off-the-shelf (COTS) software
software defined by market-driven need, commercially available and whose fitness for purpose has been demonstrated by a broad spectrum of commercial users

3.5
design authority
body responsible for the formulation of a design solution to fulfil the specified requirements and for overseeing the subsequent development and setting to work of a system in its intended environment

3.6
designer
one or more persons assigned by the Design Authority to analyse and transform specified requirements into acceptable design solutions which have the required safety integrity

3.7
element
part of a product that has been determined to be a basic unit or building block. An element may be simple or complex

3.8
error
deviation from the intended design which could result in unintended system behaviour or failure

3.9
failure
deviation from the specified performance of a system. A failure is the consequence of a fault or error in a system

3.10
fault
abnormal condition that could lead to an error or a failure in a system. A fault can be random or systematic

3.11
fault avoidance
use of design techniques which aim to avoid the introduction of faults during the design and construction of the system

3.12
fault tolerance
built-in capability of a system to provide continued correct provision of service as specified, in the presence of a limited number of hardware or software faults

3.13
firmware
ordered set of instructions and associated data stored in a way that is functionally independent of main storage, usually in a ROM

3.14
generic software
generic software is software which can be used for a variety of installations purely by the provision of application-specific data

3.15
implementer
one or more persons assigned by the Design Authority to transform specified designs into their physical realisation

3.16
product
collection of elements, interconnected to form a system, sub-system or item of equipment, in a manner which meets the specified requirements. In this European Standard, a product may be considered to consist entirely of elements of software or documentation

3.17
programmable logic controller (PLC)
solid-state control system which has a user programmable memory for storage of instructions to implement specific functions

3.18
reliability
ability of an item to perform a required function under given conditions for a given period of time

3.19
requirements traceability
objective of requirements traceability is to ensure that all requirements can be shown to have been properly met

3.20
risk
combination of the frequency, or probability, and the consequence of a specified hazardous event

3.21
safety
Freedom from unacceptable levels of risk

3.22
safety authority
body responsible for certifying that the safety-related system is fit for service and complies with relevant statutory and regulatory safety requirements

3.23
safety-related software
software which carries responsibility for safety

3.24
software
intellectual creation comprising the programs, procedures, rules and any associated documentation pertaining to the operation of a system

NOTE Software is independent of the media used for transport.

3.25
software life-cycle
activities occurring during a period of time that starts when software is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a requirements phase, development phase, test phase, integration phase, installation phase and a maintenance phase

3.26
software maintainability
capability of a system to be modified to correct faults, improve performance or other attributes, or adapt it to a different environment

3.27
software maintenance
Action, or set of actions, carried out on software after its acceptance by the final user. The aim is to improve, increase and/or correct its functionality

3.28
software safety integrity level
classification number which determines the techniques and measures that have to be applied in order to reduce residual software faults to an appropriate level

3.29
system safety integrity level
number which indicates the required degree of confidence that a system will meet its specified safety features

3.30
traceability
degree to which a relationship can be established between two or more products of a development process, especially those having a predecessor/successor or master/subordinate relationship to one another

3.31
validation
activity of demonstration, by analysis and test, that the product meets, in all respects, its specified requirements

3.32
validator
person or agent appointed to carry out validation

3.33
verification
activity of determination, by analysis and test, that the output of each phase of the life-cycle fulfils the requirements of the previous phase

3.34
verifier
person or agent appointed to carry out verification

4 Objectives and conformance

4.1 In each of the following clauses, the objectives and requirements of the clause are detailed.

4.2 To conform to this European Standard it shall be shown that each of the requirements have been satisfied to the software safety integrity level defined and therefore the clause objective has been met.

4.3 Where a requirement is qualified by the words "To the extent required by the software safety integrity level", this indicates that a range of techniques and measures can be used to satisfy that requirement.

4.4 Where 4.3 applies the tables detailed in this European Standard should be used to assist in the selection of techniques and measures appropriate to the software safety integrity level.

4.5 If a technique or measure is ranked as highly recommended (HR) in the tables then the rationale for not using that technique should be detailed and recorded either in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan. This is not necessary if an approved combination of techniques given in the corresponding table is used.

4.6 If a technique or measure is proposed to be used that is not contained in the tables then its effectiveness and suitability in meeting the particular requirement and overall objective of the clause shall be justified and recorded in either the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.

4.7 Compliance with the requirements of a particular clause and their respective techniques and measures detailed in the tables shall be assessed by the inspection of documents required by this standard, other objective evidence, auditing and the witnessing of tests.

4.8 This European Standard requires the use of a package of techniques and their correct application. These techniques are required from the tables and detailed in the bibliography.

5 Software safety integrity levels

5.1 Objective

To describe the assignment of software safety integrity levels to the software.

5.2 Requirements

5.2.1 There shall be produced, in accordance with EN 50126 and EN 50129

- System Requirements Specification,
- System Safety Requirements Specification,
- System Architecture Description,
- System Safety Plan,

which include:

- safety functions;
- configuration or architecture of the system;
- hardware reliability requirements;
- safety integrity requirements.

The software safety integrity level shall be specified through following the general process for obtaining a safety integrity level identified in EN 50126.

5.2.2 The required software safety integrity level shall be decided on the basis of the level of risk associated with the use of the software in the system and the system safety integrity level.

5.2.3 Without further precautions, the software safety integrity level shall be, as a minimum, identical to the system safety integrity level. However, if mechanisms exist to prevent the failure of a software module from causing the system to go to an unsafe state, the software safety integrity level of the module may be reduced.

5.2.4 Risks which shall be taken into account are those associated with the following hazard consequences:

- loss of human life or lives;
- injuries to or illness of persons;
- environmental pollution; and
- loss of or damage to property.

5.2.5 Risk may be quantified but it is not possible to specify the software safety integrity in the same manner. Therefore for this European Standard the software safety integrity shall be specified as one of the following five levels:

Software safety integrity level	Description of software safety integrity
4	Very High
3	High
2	Medium
1	Low
0	Non safety-related

5.2.6 The software safety integrity level shall be specified in the Software Requirements Specification (clause 8). If different software components have different software safety integrity levels, these shall be specified in the Software Architecture Specification (clause 9).

6 Personnel and responsibilities

6.1 Objective

To ensure that all personnel who have responsibilities for the software are competent to discharge those responsibilities.

6.2 Requirements

6.2.1 As a minimum, the supplier and/or developer and the customer shall implement the relevant parts of EN ISO 9001, in accordance with the guidelines contained in EN ISO 9000-3.

6.2.2 Except at software safety integrity level zero, the safety process shall be implemented under the control of an appropriate safety organisation which is compliant with the "Safety Organisation" sub-clause in the "Evidence of Safety Management" clause of EN 50129.

6.2.3 All personnel involved in all the phases of the Software Lifecycle, including management activities, shall have the appropriate training, experience and qualifications.

6.2.4 It is highly recommended that the training, experience and qualifications of all personnel involved in all the phases of the Software Lifecycle, including management activities, be justified with respect to the particular application, except at software safety integrity level zero.

6.2.5 The justification contained in 6.2.4 shall be recorded in the Software Quality Assurance Plan, and shall include evidence of competency in the following areas, as appropriate:

i) engineering appropriate to the application area;

ii) software engineering;

iii) computer-systems engineering;

iv) safety engineering;

v) legal and regulatory framework.

6.2.6 An independent assessor for the software shall be appointed. See also 6.2.10 and 14.4.1.

6.2.7 The assessor shall be given authority to perform the assessment of the software.

6.2.8 Throughout the Software Lifecycle, the parties involved shall be independent, to the extent required by the software safety integrity level, in accordance with Figure 5, which shall be interpreted as follows.

At all software safety integrity levels, the Assessor shall be approved by the Safety Authority and independent from the supplier except in the circumstances defined in 6.2.10.

The Designer/Implementer, Verifier and Validator can all belong to the same company but the following rules for minimum independence shall be complied with:

At software safety integrity level 0:

There are no constraints; the Designer/Implementer, Verifier and Validator can all be the same person.

At software safety integrity level 1 & 2:

The Verifier and Validator can be the same person but they shall not be the Designer/Implementer. However, the Designer/Implementer, Verifier and Validator can all report through the Project Manager.

At software safety integrity level 3 & 4 there are two permissible arrangements:

a) The Verifier and Validator can be the same person but they shall not also be the Designer/Implementer. In addition, the Verifier and Validator shall not all report through the Project Manager as the Designer/Implementer does and they shall have the authority to prevent the release of the product.

b) The Designer/Implementer, Verifier and Validator must all be separate persons. The Designer/Implementer and Verifier can report through the Project Manager, whereas the Validator shall not and the Validator shall have the authority to prevent the release of the product.

6.2.9 The parties responsible for the various clauses are as follows:

Software Requirements Specification (clause 8)	Designer
Software Requirements Test Specification (clause 8)	Validator
Software Architecture (clause 9)	Designer
Software Design and Development (clause 10)	Designer
Software Verification and Testing (clause 11)	Verifier
Software/Hardware Integration (clause 12)	Designer
Software Validation (clause 13)	Validator
Software Assessment (clause 14)	Assessor

6.2.10 At the discretion of the Safety Authority, the Assessor may be part of the supplier’s organisation or of the customer’s organisation but, in such cases, the Assessor shall

– be authorised by the Safety Authority,

– be totally independent from the project team,

– report directly to the Safety Authority.

7 Lifecycle issues and documentation

7.1 Objectives

7.1.1 To structure the development of the software into defined phases and activities.

7.1.2 To record all information pertinent to the software throughout the lifecycle of the software.

7.2 Requirements

7.2.1 A lifecycle model for the development of software shall be selected. It shall be detailed in the Software Quality Assurance Plan in accordance with clause 15 of this European Standard. For example, two lifecycle models are shown in Figures 3 and 4.

7.2.2 Quality Assurance procedures shall run in parallel with lifecycle activities and use the same terminology.

7.2.3 All activities to be performed during a phase shall be defined prior to the phase commencing. Each phase of the software lifecycle shall be divided into elementary tasks with a well defined input, output and activity for each of them.

7.2.4 The Software Quality Assurance Plan shall describe which verification steps and reports are required.

7.2.5 All documents shall be structured to allow continued expansion in parallel with the design process.

7.2.6 Traceability of documents shall be provided for by each document having a unique reference number and a defined and documented relationship with other documents. Each term, acronym or abbreviation shall have the same meaning in every document. If, for historical reasons, this is not possible, the different meanings shall be listed and the references given.

In addition, each document except documents for COTS software (see 9.4.5) or previously developed software (see 9.4.6) shall be written according to the following rules:

a) it shall contain or implement all applicable conditions and requirements of the predecessor document with which it has a hierarchical relationship;

b) it shall not contradict the predecessor document;

c) each term, acronym or abbreviation shall have the same meaning in every document;

d) each item or concept shall be referred to by the same name or description in every document.

7.2.7 The contents of all documents shall be recorded in a form appropriate for manipulation, processing and storage.

7.2.8 To the extent required by the software safety integrity level, the documents listed in the Documents Cross Reference Table (see below) shall be produced.

7.2.9 Depending upon the size, complexity and lifecycle of the software being developed, the number of separate documents required to be produced will vary. Some documents may be combined (providing there is no loss of the required detail in the process). For large projects it may be necessary to sub-divide the documentation listed (in a hierarchical manner) into a number of more manageable child documents. Documents which have been produced by independent teams or entities shall not be combined into a single document.

7.2.10 The relationship between the various documents identified in clause 7 can also be defined by using a DCRT (a Document Cross Reference Table). For each document listed in the "Documents" column, the phase and clause associated with its creation can be found by reading horizontally and vertically from the cell containing the symbol "█". The phases in which it is used can be found by reading vertically from the cells marked with the symbol "♦". The clause or other reference to the definition of the document can be found in the "Where defined" column. Where a clause is given, the following clauses should also be checked as they may contain further information. It should be noted that the reference for the Software Configuration Management Plan is shown in brackets because that clause simply references EN ISO 9001.

DOCUMENTS CROSS-REFERENCE TABLE

clause	8	9	10	11	12	13	14	15	16		
title	SRS	SA	SDD	SVer	S/H I	SVal	Ass	Q	Ma		
PHASES *(*) = in parallel with other phases*										**DOCUMENTS**	where defined
SYSTEM INPUTS	♦			♦	♦					System Requirements Specification	EN 50129 annex B.2.3
	♦	♦		♦	♦	♦	♦			System Safety Requirements Specification	EN 50129 annex B.2.4
	♦				♦					System Architecture Description	EN 50129 annex B.2.1
										System Safety Plan	EN 50129 EN 50126
SW PLANNING *(*)*	♦	♦	♦	♦		♦	♦	▮		Sw Quality Assurance Plan	15.4.3
						♦	♦	▮		Sw Configuration Management Plan	(15.4.2)
				▮		♦	♦			Sw Verification Plan	11.4.1
				▮		♦	♦			Sw Integration Test Plan	11.4.5
					▮	♦	♦			Sw/Hw Integration Test Plan	12.4.1
						▮	♦			Sw Validation Plan	13.4.3
							♦		▮	Sw Maintenance Plan	16.4.3
										Data Preparation Plan	17.4.2.1
										Data Test Plan	17.4.2.4
SW REQUIREMENTS	▮	♦	♦	♦	♦	♦	♦			Sw Requirements Specification	8.4.1
										Application Requirements Specification	17.4.1.1
	▮			♦	♦	♦	♦			Sw Requirements Test Specification	8.4.13
				▮						Sw Requirements Verification Report	11.4.11
SW DESIGN		▮	♦	♦	♦	♦	♦			Sw Architecture Specification	9.4.1
			▮	♦	♦	♦	♦			Sw Design Specification	10.4.3
				▮						Sw Arch. and Design Verification Report	11.4.12
SW MODULE DESIGN			▮	♦	♦	♦	♦			Sw Module Design Specification	10.4.3
			▮	♦	♦	♦	♦			Sw Module Test Specification	10.4.14
				▮						Sw Module Verification Report	11.4.13
CODE			▮	♦	♦	♦	♦			Sw Source Code	
				▮		♦	♦			Sw Source Code Verification Report	11.4.14
MODULE TESTING			▮	♦						Sw Module Test Report	10.4.14
SW INTEGRATION				▮						Sw Integration Test Report	11.4.15
										Data Test Report	17.4.2.4
SW/HW INTEGRATION					▮					Sw/Hw Integration Test Report	12.4.8
VALIDATION *(*)*						▮				Sw Validation Report	13.4.10
ASSESSMENT *(*)*							▮			Sw Assesment Report	14.4.9
MAINTENANCE									▮	Sw Change Records	16.4.9
									▮	Sw Maintenance Records	16.4.8

8 Software requirements specification

8.1 Objectives

8.1.1 To describe a document which defines a complete set of requirements for the software meeting all System Requirements to the extent required by the Software safety Integrity level. It serves the purpose of a comprehensive document for each software engineer and makes it unnecessary for him to screen for requirements in any other document.

8.1.2 To describe the Software Requirements Test Specification.

8.2 Input documents

1) System Requirements Specification

2) System Safety Requirements Specification

3) System Architecture Description

4) Software Quality Assurance Plan

8.3 Output documents

1) Software Requirements Specification

2) Software Requirements Test Specification

8.4 Requirements

8.4.1 The Software Requirements Specification shall express the required properties of the software being developed, not the procedures to develop them. These properties, which are all (except safety) defined in ISO/IEC 9126, shall include :

– functionality (including capacity and response time performance);

– reliability and maintainability;

– safety (including safety functions and their associated software safety integrity levels);

– efficiency;

– usability;

– portability.

The software safety integrity level shall be derived as defined in clause 5 and recorded in the Software Requirements Specification.

8.4.2 To the extent required by the software safety integrity level the Software Requirements Specification shall be expressed and structured in such a way that it is

i) complete, clear precise, unequivocal, verifiable, testable, maintainable and feasible,

ii) traceable back to all documents mentioned under 8.2.

8.4.3 The Software Requirements Specification shall include modes of expression and descriptions which are understandable to the responsible personnel involved in the whole life cycle of the system.

8.4.4 The Software Requirements Specification shall identify and document all interfaces with any other systems, either within or outside the equipment under control, including operators, wherever a direct connection exists or is planned.

8.4.5 All relevant modes of operation shall be detailed in the Software Requirements Specification.

8.4.6 All relevant modes of behaviour of the programmable electronics, in particular failure behaviour, shall be detailed in the Software Requirements Specification.

8.4.7 Any constraints between the hardware and the software shall be identified and documented in the Software Requirements Specification.

8.4.8 The Software Requirements Specification shall indicate the degree of software self-checking and the specified degree of hardware checking by the software. Software self-checking consists of the detection and reporting by the software of its own failures and errors.

8.4.9 The Software Requirements Specification shall include requirements for the periodic testing of functions to the extent required by the System Safety Requirements Specification.

8.4.10 The Software Requirements Specification shall include requirements to enable all safety functions to be testable during overall system operation to the extent required by the System Safety Requirements Specification.

8.4.11 When the software is required to perform functions especially those related to achieving the required system safety integrity level, then these shall be clearly identified in the Software Requirements Specification.

8.4.12 When the software is required to perform non-safety functions then these shall be clearly identified in the Software Requirements Specification.

8.4.13 A Software Requirements Test Specification shall be developed from the Software Requirements Specification. This test specification shall be used for verification of all the requirements as described in the Software Requirements Specification and also as a description of the tests to be performed on the completed software.

8.4.14 The Software Requirements Test Specification shall identify for each required function the test cases including:

i) the required input signals with their sequences and their values;

ii) the anticipated output signals with their sequences and their values;

iii) the acceptance criteria, including performance and quality aspects.

8.4.15 Traceability to requirements shall be an important consideration in the validation of a safety-related system and means shall be provided to allow this to be demonstrated throughout all phases of the lifecycle.

8.4.16 Any untraceable material shall be shown to have no bearing upon the safety or integrity of the system.

9 Software architecture

9.1 Objectives

9.1.1 To develop a software architecture that achieves the requirements of the Software Requirements Specification to the extent required by the software safety integrity level.

9.1.2 To review the requirements placed on the software by the system architecture.

9.1.3 To identify and evaluate the significance of Hardware/Software interactions for safety.

9.1.4 To choose a design method if one has not been previously defined.

9.2 Input documents

1) Software Requirements Specification

2) System Safety Requirements Specification

3) System Architecture Description

4) Software Quality Assurance Plan

9.3 Output documents

Software Architecture Specification

9.4 Requirements

9.4.1 The proposed software architecture shall be established by the software supplier and/or developer and detailed in the Software Architecture Specification.

9.4.2 The Software Architecture Specification shall consider the feasibility of achieving the Software Requirements Specification at the required software safety integrity level.

9.4.3 The Software Architecture Specification shall identify, evaluate and detail the significance of all hardware/software interactions. As required by EN 50126 and EN 50129, the preliminary studies concerning the interactions between hardware and software shall have been recorded in the System Safety Requirements Specification.

9.4.4 The Software Architecture Specification shall identify all software components and for these components identify:

i) whether these components are new, existing or proprietary;

ii) whether these components have been previously validated and if so their validation conditions;

iii) the software safety integrity level of the component.

9.4.5 The use of COTS software shall be subject to the following restrictions:

i) for software safety integrity level 0, the use of the COTS software shall be accepted with no further precautions;

ii) if COTS software is to be used at software safety integrity levels 1 or 2, it shall be included in the software validation process;

iii) if COTS software is to be used at software safety integrity levels 3 or 4, the following precautions shall also be taken:

   a) the COTS software shall be included in the validation testing;

   b) an analysis of possible failures shall be carried out;

c) a strategy shall be defined to detect failures of the COTS software and to protect the system from these failures;

d) the protection strategy shall be the subject of validation testing;

e) error logs shall exist and shall be evaluated;

f) as far as practicable, only the simplest functions of the COTS software shall be used.

9.4.6 If previously developed software is to be used as part of the design then it shall be clearly identified and documented. The Software Architecture Specification shall justify the software's suitability in satisfying the Software Requirements Specification and the software safety integrity level. The effects of any changes to the software on the rest of the system must be carefully considered in order to decide whether they require a re-inspection and re-assessment. There shall be evidence that interface specifications to other modules which are not being re-verified, re-validated and re-assessed are being followed.

9.4.7 Whenever possible existing verified software modules developed according to this standard shall be used in the design.

9.4.8 The Software Architecture shall minimise the safety part of the application.

9.4.9 Where the software consists of components of different software safety integrity levels then all of the software components shall be treated as belonging to the highest software safety integrity level unless there is evidence of independence between the higher software safety integrity level components and the lower software safety integrity level components. This evidence shall be recorded in the Software Architecture Specification.

9.4.10 The Software Architecture Specification shall identify the strategy for the software development to the extent required by the software safety integrity level. The Software Architecture Specification shall be expressed and structured in such a way that it is

i) complete, clear, precise, unequivocal, verifiable, testable, maintainable and feasible,

ii) traceable back to the Software Requirements Specification.

9.4.11 The Software Architecture Specification shall justify the balance taken between the strategies of avoiding faults and handling faults.

9.4.12 The Software Architecture Specification shall justify that the techniques and measures chosen form a set which satisfies the Software Requirements Specification at the required software safety integrity level.

10 Software design and implementation

10.1 Objectives

10.1.1 To design and implement software of a defined software safety integrity level from the Software Requirements Specification and the Software Architecture Specification.

10.1.2 To achieve software which is analysable, testable, verifiable and maintainable. Module testing is also included in this phase. As verification and test will be a critical element in the validation, particular consideration shall be given to verification and test needs throughout the design and development, in order to ensure the resultant system and its software will be readily testable from the outset.

10.1.3 To select a suitable set of tools, including languages and compilers, for the required software safety integrity level, over the whole lifecycle of the software which assists verification, validation, assessment and maintenance.

10.1.4 To carry out software integration.

10.2 Input documents

1) Software Requirements Specification
2) Software Architecture Specification
3) Software Quality Assurance Plan

10.3 Output documents

1) Software Design Specification
2) Software Module Design Specification
3) Software Module Test Specification
4) Software Source code and supporting documentation
5) Software Module Test Report

10.4 Requirements

10.4.1 The Software Requirements Specification and the Software Architecture Specification shall be available, although not necessarily finalised, prior to the start of the design process.

10.4.2 The size and complexity of the software developed shall be kept to a minimum.

10.4.3 The Software Design Specification shall describe the software design based on a decomposition into modules with each module having a Software Module Design Specification and a Software Module Test Specification.

10.4.4 The Software Design Specification shall address:

i) software components traced back to software architecture and their safety integrity level,

ii) interfaces of software components with the environment,

iii) interfaces between the software components,

iv) data structure,

v) partitioning of requirements on components,

vi) main algorithms and sequencing,

vii) diagrams.

10.4.5 The Software Module Design Specifications shall address (one Software module Design Specification per module):

i) identification of all lowest software components (called modules in the present standard) traced back to the upper level;

ii) their detailed interfaces with environment and other modules with detailed inputs and outputs;

iii) their safety integrity level;

iv) detailed algorithms and data structures.

Each Software Module design Specification should be self consistent and allow coding of the corresponding module.

10.4.6 Each software module shall be readable, understandable and testable.

10.4.7 A suitable set of tools, including design methods, languages and compilers shall be selected for the required software safety integrity level over the whole lifecycle of the software.

10.4.8 When applicable, automatic testing tools and integrated development tools shall be used. This shall take account of the needs of the Verifier and Validator.

10.4.9 To the extent required by the software safety integrity level, the programming language selected shall have a translator/compiler which has one of the following:

i) a "Certificate of Validation" to a recognised National/International standard;

ii) an assessment report which details its fitness for purpose;

iii) a redundant signature control based process that provides detection of the translation errors.

10.4.10 The language chosen shall meet the following requirements:

i) the language chosen shall contain features that facilitate the identification of programming errors;

ii) the language chosen shall support features that match the design method.

10.4.11 When 10.4.10 cannot be satisfied then a justification for any alternative language detailing its fitness for purpose shall be recorded in the Software Architecture Specification or Software Quality Assurance Plan.

10.4.12 Coding standards shall be developed and used for the development of all software. These shall be referenced in the Software Quality Assurance Plan (see 15.4.5).

10.4.13 The coding standards shall specify good programming practice, proscribe unsafe language features and describe procedures for source code documentation. As a minimum, each software module shall contain in the source code the information defined in the following (non-exhaustive) list:

– author;

– configuration history;

– short description.

The use of a standard form for this information is recommended. It should be the same for all the modules.

10.4.14 Software Module Testing: Each module shall have a Software Module Test Specification which the module shall be tested against. These tests shall show that each module performs its intended function. The Software Module Test Specification shall define the required degree of test coverage.

A Software Module Test Report shall be produced and shall include the following features:

i) a statement of the test results and whether each module has met the requirements of its Software Module Design Specification;

ii) a statement of test coverage shall be provided for each module, showing that all source code instructions have been executed at least once;

iii) it shall be in a form that is auditable;

iv) test cases and their results shall be recorded in a machine readable form for subsequent analysis; Tests should be repeatable and be performed by automatic means, if practicable.

Checking that the module has correctly satisfied its test specification is a verification activity, see clause 11.

10.4.15 In accordance with the required software safety integrity level the design method chosen shall possess features that facilitate:

i) abstraction, modularity and other features which control complexity;

ii) the clear and precise expression of

- functionality,
- information flow between components,
- sequencing and time related information,
- concurrency,
- data structure and properties;

iii) human comprehension;

iv) verification and validation.

10.4.16 The design method chosen shall possess features that facilitate software maintenance. Such features shall include modularity, information hiding and encapsulation.

10.4.17 The integration of software modules shall be the process of progressively combining individual and previously tested modules of software into a composite whole (or into a number of composite sub-systems) in order that the module interfaces and the assembled software may be adequately proven prior to system integration and test.

10.4.18 Within the context of this standard, and to a degree appropriate to the specified software safety integrity level, traceability shall particularly address:

i) traceability of requirements to the design or other objects which fulfil them;

ii) traceability of design objects to the implementation objects which instantiate them.

The output of the traceability process shall be the subject of formal configuration management.

11 Software verification and testing

11.1 Objective

To the extent required by the software safety integrity level, to test and evaluate the products of a given phase to ensure correctness and consistency with respect to the products and standards provided as input to that phase.

11.2 Input documents

1) System Requirements Specification
2) System Safety Requirements Specification
3) Software Requirements Specification
4) Software Requirements Test Specification
5) Software Architecture Specification
6) Software Design Specification
7) Software Module Design Specification
8) Software Module Test Specification
9) Software Source code and supporting documentation
10) Software Quality Assurance Plan
11) Software Module Test Report

11.3 Output documents

1) Software Verification Plan
2) Software Requirements Verification Report
3) Software Architecture and Design Verification Report
4) Software Module Verification Report
5) Software Source Code Verification Report
6) Software Integration Test Plan
7) Software Integration Test Report

11.4 Requirements

11.4.1 A Software Verification Plan shall be created in order that verification activities may be properly directed and that particular design or other verification needs may be suitably provided for. During development (and depending upon the size of the system) the plan may be sub-divided into a number of child documents and be naturally added to, as the detailed needs of verification become clearer.

11.4.2 The Software Verification Plan shall document all the criteria, techniques and tools to be utilised in the verification process for that phase.

11.4.3 The Software Verification Plan shall describe the activities to be performed to ensure correctness and consistency with respect to the products and standards provided as input to that phase.

11.4.4 The Software Verification Plan shall address the following:

i) the selection of verification strategies and techniques. To avoid undue complexity in the assessment of the verification and test activity, preference should be given to the selection of test cases and methods etc., which are in themselves readily analysable;

ii) the selection and utilisation of the software test equipment;

iii) the selection and documentation of verification activities;

iv) the evaluation of verification results gained;

v) the evaluation of the reliability requirements;

vi) the roles and responsibilities of those involved in the test process;

vii) the degree of test coverage required to be achieved.

11.4.5 The Software Integration Test Plan shall document the following:

i) test cases and test data;

ii) types of tests to be performed;

iii) test environment, tools, configuration and programs;

iv) test criteria on which the completion of the test will be judged.

11.4.6 In each development phase it shall be shown that the functional, reliability, performance and safety requirements are met.

11.4.7 Verification shall be carried out by an independent party to the extent required by the software safety integrity level as defined in Figure 5.

11.4.8 Testing which is not fully documented and is performed by the designer prior to verification shall not be regarded as part of the verification.

11.4.9 The results of each verification shall be retained in a form defined or referenced in the Software Verification Plan such that it is auditable.

11.4.10 After each verification activity a verification report shall be produced stating either that the software has passed the verification or the reasons for the failures. The verification reports shall address the following:

i) items which do not conform to the Software Requirements Specification, Software Design Specification or Software Module Design Specifications;

ii) items which do not conform to the Software Quality Assurance Plan;

iii) modules, data, structures and algorithms poorly adapted to the problem;

iv) detected errors or deficiencies;

v) the identity and configuration of the items verified.

11.4.11 Software Requirements Verification**:** Once the Software Requirements Specification has been established, verification shall address:

i) the adequacy of the Software Requirements Specification in fulfilling the requirements set out in the System Requirements Specification, the System Safety Requirements Specification and the Software Quality Assurance Plan;

ii) the adequacy of the Software Requirements Test Specification as a test of the Software Requirements Specification;

iii) the internal consistency of the Software Requirements Specification.

The results shall be recorded in a Software Requirements Verification Report.

11.4.12 Software Architecture and Design Verification**:** After the Software Architecture Specification and the Software Design Specification have been established, verification shall address:

i) the adequacy of the Software Architecture Specification and the Software Design Specification in fulfilling the Software Requirements Specification;

ii) the adequacy of the Software Design Specification for the Software Requirements Specification with respect to consistency and completeness;

iii) the adequacy of the Software Integration Test Plan as a set of test cases for the Software Architecture Specification and the Software Design Specification;

iv) the internal consistency of the Software Architecture and Design Specifications.

The results shall be recorded in a Software Architecture and Design Verification Report.

11.4.13 Software Module Verification: After each Software Module Design Specification has been established, verification shall address:

i) the adequacy of the Software Module Design Specification in fulfilling the Software Design Specification;

ii) the adequacy of the Software Module Test Specification as a set of test cases for the Software Module Design Specification;

iii) the decomposition of the Software Design Specification into software modules and the Software Module Design Specifications with reference to

- feasibility of the performance required,
- testability for further verification,
- readability by the development and verification team, and
- maintainability to permit further evolution;

iv) the adequacy of the Software Module Test Reports as a record of the tests carried out in accordance with the Software Module Test Specification.

The results shall be recorded in a Software Module Verification Report.

11.4.14 Software Source Code Verification: To the extent demanded by the software safety integrity level the Software Source Code shall be verified to ensure conformance to the Software Module Design Specification and the Software Quality Assurance Plan. This shall include a check to determine whether the coding standards have been applied correctly.

The results shall be recorded in a Software Source Code Verification Report.

11.4.15 A Software Integration Test Report shall be produced as follows:

i) a Software Integration Test Report shall be produced stating the test results and whether the objectives and criteria of the Software Integration Test Plan have been met. If there is a failure, the reasons for the failure shall be recorded;

ii) the Software Integration Test Report shall be in a form that is auditable;

iii) test cases and their results shall be recorded, preferably in machine readable form for subsequent analysis;

iv) tests should be repeatable and be performed by automatic means,if practicable;

v) the identity and configuration of the items verified.

11.4.16 For software/hardware integration, see 12.4.8.

12 Software/hardware integration

12.1 Objectives

12.1.1 To demonstrate that the software and the hardware interact correctly to perform their intended functions.

12.1.2 To combine the software and hardware, ensuring their compatibility, to meet the System Safety Requirements Specification and the requirements of the intended software safety integrity level.

12.2 Input documents

1) System Requirements Specification

2) System Safety Requirements Specification

3) System Architecture Description

4) Software Requirements Specification

5) Software Requirements Test Specification

6) Software Architecture Specification

7) Software Design Specification

8) Software Module Design Specification

9) Software Module Test Specification

10) Software Source code and supporting documentation

11) Hardware documentation

12.3 Output documents

1) Software/Hardware Integration Test Plan

2) Software/Hardware Integration Test Report

12.4 Requirements

12.4.1 For software safety integrity levels greater than zero, a Software/Hardware Integration Test Plan will be created early in the development lifecycle, in order that integration activities may be properly directed and that particular design or other integration needs may be suitably provided for. During development (and depending upon the size of the system) the plan may be subdivided into a number of child documents and be naturally added to, as the hardware and software designs evolve and the detailed needs of integration become clearer.

12.4.2 The Software/Hardware Integration Test Plan shall document the following:

i) test cases and test data;

ii) types of tests to be performed;

iii) test environment including tools, support software and configuration description; and

iv) test criteria on which the completion of the test will be judged.

12.4.3 The Software/Hardware Integration Test Plan shall distinguish between those activities which can be carried out by the developer on his premises and those that require access to the user's site.

12.4.4 The Software/Hardware Integration Test Plan shall distinguish between the following activities:

i) merging of software onto the target hardware;

ii) system integration;

12.4.5 Tools and facilities identified in the Software/Hardware Integration Test Plan should be available at the earliest practicable time.

12.4.6 During Software/Hardware Integration any modification or change to the integrated system shall be subject to an impact study which shall identify all modules impacted and the necessary reverification activities.

12.4.7 Test cases and their results shall be recorded, preferably in machine readable form for subsequent analysis.

12.4.8 A Software/Hardware Integration Test Report shall be produced as follows:

i) Software/Hardware Integration Test Report shall state the test results and whether the objectives and criteria of the Software/Hardware Integration Test Plan have been met. If there is a failure, it shall be recorded;

ii) the Software/Hardware Integration Test Report shall be in a form that is auditable;

iii) test cases and their results shall be recorded, preferably in amachine-readable form for subsequent analysis;

iv) the Software/Hardware Integration Test Report shall also contain evidence that the Verifier is satisfied with the adequacy of the Software/Hardware Integration Test Report as a record of the tests carried out in accordance with the Software/Hardware Integration Test Plan;

v) the Software/Hardware Integration Test Report shall document the identity and configuration of all items involved.

13 Software validation

13.1 Objective

To analyse and test the integrated system to ensure compliance with the Software Requirements Specification with particular emphasis on the functional and safety aspects according to the Software Safety integrity level.

13.2 Input documents

1) Software Requirements Specification

2) All Hardware and Software Documentation

3) System Safety Requirements Specification

13.3 Output documents

1) Software Validation Plan

2) Software Validation Report

13.4 Requirements

13.4.1 Analysing and testing shall be the main validation activity.

13.4.2 Simulation and modelling may be used to supplement the validation process.

13.4.3 A Software Validation Plan shall be established and detailed in suitable documentation.

13.4.4 The Software Validation Plan shall be developed, performed and results evaluated by an independent party to the extent required by the software safety integrity level.

13.4.5 If required by the software safety integrity level, the scope and contents of the Software Validation Plan shall be agreed with the assessor. This agreement shall also make a statement concerning the presence of the assessor during testing.

13.4.6 The Software Validation Plan shall include a summary justifying the validation strategy chosen. The justification shall include consideration, according to the required software safety integrity level, of:

i) manual or automated techniques or both;

ii) static or dynamic techniques or both;

iii) analytical or statistical techniques or both.

13.4.7 The Software Validation Plan shall identify the steps necessary to demonstrate the adequacy of the:

– Software Requirements Specification;

– Software Architecture Specification;

– Software Design Specification;

– Software Module Design Specification;

in fulfilling the safety requirements set out in the System Safety Requirements Specification. The Validator shall check that the verification process is complete.

13.4.8 Measurement equipment used for validation shall be calibrated appropriately. Any tools, hardware or software, used for validation shall be shown to be suitable for the purpose.

13.4.9 The software shall be exercised by simulation of input signals present during normal operation, anticipated occurrences and undesired conditions requiring system action.

13.4.10 The results of the validation shall be documented in the Software Validation Report in an auditable form.

13.4.11 Once hardware/software integration is finished, a Software Validation Report shall be produced as follows:

i) it shall state whether the objectives and criteria of the Software Validation Plan have been met. If there is a failure, it shall be recorded;

ii) it shall state the tests results and whether the whole software on its target machine fulfils the requirements set out in the Software Requirements Specification;

iii) an evaluation of the test coverage on the requirements of the Software Requirements Specification shall be provided;

iv) the Software Validation Report shall be in a form that is auditable;

v) test cases and their results shall be recorded in a machine readable form for subsequent analysis;

vi) tests should be repeatable and be performed by automatic means, if practicable.

13.4.12 The Software Validation Report shall document the identity and configuration of all:

i) the hardware and software used;

ii) the equipment used;

iii) the equipment's calibration;

iv) the simulation models used;

v) the discrepancies found;

vi) the corrective actions performed.

13.4.13 Any discrepancies found, including detected errors, shall be clearly identified in a separate section of the Software Validation Report and included in any release note which accompanies the delivered software.

13.4.14 The software shall be tested against the Software Requirements Test Specification. These tests shall show that all of the requirements in the Software Requirements Specification are correctly performed.

The results shall be recorded in a Software Validation Report.

14 Software assessment

14.1 Objective

To evaluate that the lifecycle processes and products resulting are such that the software is of the defined software safety integrity level and is fit for its intended application.

14.2 Input documents

1) System Safety Requirements Specification

2) All Hardware and Software Documentation

14.3 Output documents

Software Assessment Report

14.4 Requirements

14.4.1 For software of safety integrity level zero:

i) the Assessor shall only be required to confirm that this is the appropriate software safety integrity level;

ii) it is also possible to agree this level between the supplier and the user at the time of tendering.

14.4.2 Software with a Software Assessment Report from another Assessor does not have to be an object for an entirely new assessment. The second Assessor shall check that the software is of the required software safety integrity level and that it is fit for its intended application on the intended target computer.

14.4.3 The Assessor shall have access to the design and development process and all project related documentation.

14.4.4 The assessment of the software shall be carried out by an Assessor who is independent from the design team.

14.4.5 The Assessor shall assess that the software of the system is fit for its intended purpose and responds correctly to safety issues derived from the System Safety Requirements Specification.

14.4.6 To the extent required by the software safety integrity level, the Assessor shall decide if appropriate methods have been selected and applied at each phase of the software lifecycle.

14.4.7 If required by the software safety integrity level, the Assessor shall agree the scope and contents of the Software Validation Plan. This agreement shall also make a statement concerning the presence of the Assessor during testing.

14.4.8 The Assessor may ask for additional verification and validation work if he so chooses.

14.4.9 The Assessor shall produce a report for each review which shall detail his assessment results.

14.4.10 If, in the opinion of the Assessor, the software is fit for its intended application, the final Software Assessment Report shall include a statement as to the software safety integrity level achieved by the software.

14.4.11 When the software is not fit for its purpose or has not achieved the required software safety integrity level then the Assessor shall only report the non-conformities in the Software Assessment Report and shall not give any technical solution.

15 Software quality assurance

15.1 Objectives

15.1.1 To identify, monitor and control all those activities, both technical and managerial, which are necessary to ensure that the software achieves the quality required. This is necessary to provide the required qualitative defence against systematic faults and to ensure that an audit trail can be established to allow verification and validation activities to be undertaken effectively.

15.1.2 To provide evidence that the above activities have been carried out.

15.2 Input documents

All the documents available at each stage of the lifecycle

15.3 Output documents

1) Software Quality Assurance Plan

2) Software Configuration Management Plan

All the above plans shall be issued at the beginning of the project and updated during the lifecycle.

15.4 Requirements

15.4.1 The supplier and/or developer shall have and use as a minimum a Quality Assurance System compliant with EN ISO 9000 series, to support the requirements of this European Standard. EN ISO 9001 accreditation is highly recommended.

15.4.2 As a minimum, the supplier and/or developer and the customer shall implement for the software development the relevant parts of EN ISO 9001, in accordance with the guidelines contained in EN ISO 9000-3.

15.4.3 The supplier and/or developer shall prepare and document, on a project by project basis, a Software Quality Assurance Plan to implement the requirements of 15.4.1 and 15.4.2 of this European Standard, which shall be expressed in measurable terms wherever possible.

15.4.4 The Software Quality Assurance Plan shall have a paragraph specifying details about its own updating throughout the project: frequency, responsibility, method.

15.4.5 All activities, actions, documents, etc. required by all the sections of EN ISO 9000-3 and of this European Standard (annex A included) shall be specified or referenced in the Software Quality Assurance Plan and tailored to the specific development. None of the lists in EN ISO 9000-3 shall be presumed to be exhaustive.

As a minimum, except at software safety integrity level zero, the following items shall also be specified or referenced in the Software Quality Assurance Plan.

This is to ensure that all the safety aspects in the Software with respect to the required Software Safety integrity level will be covered.

The present list is not exhaustive:

i) definition of the life-cycle model
definition of each phase including:

- activities and elementary tasks;
- entry and exit criteria;
- inputs and outputs of each phase;
- major quality activities in each phase;
- organisational unit responsible for each activity and elementary task;

ii) requirements traceability;

iii) documentation structure traceability;

iv) documentation associated with the development, verification and validation, operation and maintenance of software;

v) system integration procedures;

vi) coding standards to be used;

vii) assessment of previous validation tests;

viii) the definition of the metrics (quantitative measures) to be carried out on both the product and the process. For the software product metrics carried out, reference shall be made to the quality characteristics and evaluation guidelines defined by ISO/IEC 9126.

15.4.6 As a minimum, configuration management shall be carried out in accordance with the guidelines contained in EN ISO 9000-3.

Each software document shall be placed under configuration control before the release of its first approved version. The software source code shall be placed under configuration control before the commencement of documented module testing.

It shall not be possible to make any unauthorised changes to any item under Configuration Management Control. Precautions shall be taken to prevent or detect errors occurring in machine readable code during storage, transfer, transmission or duplication.

Configuration Management shall not be limited to the strict product development and maintenance, but it shall also cover the environment used during the full lifecycle. This extension, necessary for the reproducibility of the development and for the maintenance activities, shall include computer configuration files, assemblers, compilers, debuggers and all the other used tools.

15.4.7 The adequacy and results of Software Verification Plans shall be examined.

15.4.8 The supplier and/or developer shall establish, document and maintain procedures for External Supplier Control, including:

- methods to ensure that software provided by external suppliers adheres to established requirements. Previously developed software shall be assured to be compliant with the required software safety integrity level and dependability. New software shall be developed and maintained in conformity with the Software Quality Assurance Plan of the Supplier or with a specific Software Quality Assurance Plan prepared by the external supplier in accordance with the Software Quality Assurance Plan of the Supplier;
- methods to ensure that the requirements provided to the External Software Supplier are adequate and complete.

15.4.9 The supplier and/or developer shall establish, document and maintain procedures for Problem Reporting and Corrective Actions. These procedures, as part of the Quality Assurance System, shall implement the relevant parts of EN ISO 9001, especially covering at least the following aspects:

- define the documentation needed for problem reporting and/or corrective actions, with the aim of giving feedback to the responsible management;
- define analysis of the information collected in the problem reports to identify its causes;
- define the practices to be followed for reporting, tracking and resolving problems identified both during the development phase and during software maintenance;
- define preventive actions to deal with problems to a level corresponding to the required software safety integrity level;
- define the specific organisational responsibilities with regard to development and software maintenance;
- define how to apply controls to ensure that corrective actions are taken and that they are effective;
- define the forms to be used;
- define the requirements for re-test, re-verification, re-validation and re-assessment.

As a minimum, problem reporting and corrective action management shall be applied in the software lifecycle starting immediately after Software Integration and before the starting of formal Software Validation, also covering the whole phase of Software Maintenance.

16 Software maintenance

16.1 Objective

To ensure that the software performs as required, preserving the required software safety integrity level and dependability when making corrections, enhancements or adaptations to the software itself.

16.2 Input documents

All documents

16.3 Output documents

1) Software Maintenance Plan

2) Software Change Records

3) Software Maintenance Record

16.4 Requirements

16.4.1 As a minimum, maintenance shall be carried out in accordance with the guidelines contained in EN ISO 9000-3.

In addition, the following requirements concerning software maintenance shall also be met.

16.4.2 Maintainability shall be designed into the software system, in particular, by following the requirements of clause 10 of this European Standard. ISO/IEC 9126 should also be employed in order to require and verify a minimum level of maintainability.

16.4.3 Procedures for the maintenance of software shall be established and recorded in the Software Maintenance Plan. These procedures shall also include:

i) control of error reporting, error logs, maintenance records, change authorisation and software/system configuration;

ii) verification, validation and assessment; and

iii) definition of the Authority which approves the changed software.

16.4.4 The maintenance activities shall be audited against the Software Maintenance Plan, at intervals defined in the Software Quality Assurance Plan.

16.4.5 Maintenance shall be performed with the same level of expertise, tools, documentation, planning and management as the initial development of the system. This shall apply also to configuration management, change control, document control, and independence of involved parties.

16.4.6 This European Standard is not intended to be retrospective. It therefore applies primarily to new developments and only applies in its entirety to existing systems if these are subjected to major modifications. For software safety integrity level 3 or 4, the contracting entities shall, before starting work on any change, decide whether the maintenance actions are to be considered as major or minor or whether the existing maintenance methods for the system are adequate. For software safety integrity levels 0, 1 or 2, the same decision shall be taken by the supplier.

16.4.7 External supplier control, problem reporting and corrective actions shall be managed with the same criteria specified in the relevant paragraphs of the Software Quality Assurance clause.

16.4.8 A Software Maintenance Record shall be established for each Software Item before its first release, and it shall be maintained. In addition to the requirements of EN ISO 9000-3 for "Maintenance Records and Reports", this Record shall also include:

i) references to all the Software Change Records for that Software Item;

ii) change consequence information;

iii) test cases for components, including revalidation and regression testing data; and

iv) software configuration history.

16.4.9 A Software Change Record shall be established for each maintenance activity. This record shall include:

i) the modification or change request;

ii) an analysis of the impact of the maintenance activity on the overall system, including hardware, software, human interaction and the environment and possible interactions;

iii) the detailed specification of the modification or change; and

iv) revalidation, regression testing and re-assessment of the modification or change to the extent required by the software safety integrity level. The responsibility for revalidation can vary from project to project, according to the software safety integrity level. Also the impact of the modification or change on the process of revalidation can be confined to different system levels (only changed modules, all identified affected modules, the complete system). Therefore the Software Validation Plan shall address both problems, according to the software safety integrity level. The degree of independence of revalidation shall be the same as that for validation.

17 Systems configured by application data

17.1 Objectives

17.1.1 A characteristic feature of railway control and protection systems is the need to design each installation to meet the individual requirements for a specific application. A system configured by application data allows type-approved generic software to be used, with the individual requirements for each installation defined as data (application specific data). This data is normally in the form of tabular information or an application specific language which is interpreted by the generic software.

17.1.2 For a safety critical system, the high level of resources required to develop software to achieve the required system safety integrity level for the system makes the adoption of a system configured by application data very attractive because it allows the re-use of generic software. However, as the safe operation of the system is likely to depend on the correctness of the data, the procedures used for development of the data must also be to an appropriate system safety integrity level.

17.1.3 The sections below describe the requirements of this European Standard for the initial development of the generic software for a system configured by application data, and for the subsequent development of each set of installation-specific data.

17.2 Input documents

1) Software Requirements Specification

2) Software Architecture Specification

17.3 Output documents

1) Application Requirements Specification

2) Data Preparation Plan

3) Data Test Plan

4) Data Test Report

17.4 Requirements

17.4.1 Data Preparation Lifecycle

Figure 6 illustrates a lifecycle for an application making use of hardware and generic software, together with application-specific data. The phases of the lifecycle are as follows.

17.4.1.1 Application Requirements Specification

The requirements for the application shall be defined. This shall include requirements which are specific to the individual installation (e.g. track layout, signal locations, speed limits), and standards which the application must comply with (e.g. signalling principles, system safety integrity levels).

17.4.1.2 Overall Installation Design

The system architecture shall be defined, and the quantity and type of the generic components to be used shall be specified. Those components which contain software shall have been developed in conformance with this European Standard. The functions specified in the requirements shall be allocated to the components, and the physical location of each component shall be defined.

17.4.1.3 Data Preparation

The data preparation process shall include the production of specific information (e.g. control tables), production of the data source code and its compilation, checking and other verification activities, and testing of the application data.

17.4.1.4 Integration and Acceptance

For some systems the application data will be integrated with the generic hardware and software for a factory test before installation on site. This may not be necessary where a sufficient degree of confidence can be obtained by other means. The equipment shall then be installed on site, and integration tests carried out on the new equipment. Finally the system shall be commissioned as a fully operational system, and a final acceptance process shall be carried out on the complete installation.

17.4.1.5 Validation and Assessment

Validation and assessment activities shall audit the performance of each stage of the life-cycle.

17.4.2 Data Preparation Procedures and Tools

For each new type of system configured by application data, specific data preparation procedures and tools shall be developed to allow the data preparation lifecycle specified in 17.4.1 to be applied to installations of the new system. Development of these procedures and tools shall be carried out in accordance with this European Standard in parallel with the generic software and hardware for the system. The verification, validation and assessment activities shall ensure that the data preparation tools and the generic software are compatible.

17.4.2.1 At the Software Design phase for the system configured by application data, a Data Preparation Plan shall be produced to define a documentation structure for the data preparation process. These documents shall be related to the data preparation lifecycle model described in 17.4.1. The plan shall specify the data preparation procedures and tools to be used to meet the required system safety integrity levels. The plan shall specify the requirements for the independence between staff carrying out verification, validation and design tasks.

17.4.2.2 The Data Preparation Plan shall allocate a safety integrity level to any hardware or software tools used in the data preparation lifecycle. This safety integrity level shall be derived from the required system safety integrity level and the degree to which the output of each tool is checked by other manual or automated procedures.

17.4.2.3 Where possible the Data Preparation Plan shall call for notations for specifying requirements and design which are familiar to applications engineers, e.g. standard signalling plans and control tables. Where new notations are introduced, the necessary user documentation must be provided and training shall also be provided where appropriate.

17.4.2.4 The verification, test, validation and assessment reports required to demonstrate that the data preparation has been carried out in accordance with the plan can be standardised in the form of checklists to minimise the workload in producing documentation for each installation. This information shall be contained in the Data Test Plan and the results shall be recorded in the Data Test Report.

17.4.2.5 All data and associated documentation shall be subject to the configuration management requirements of section 15 of this standard. The configuration management records shall list the version of generic software with which the data has been designed to operate, and the versions of the tools used in the data preparation process.

17.4.3 Software Development

Development of the generic software to be used in a system configured by application data shall comply with the requirements in clauses 1 to 16 of this standard. The following additional requirements shall also be observed.

17.4.3.1 During the Software Requirements Specification phase, those functions which make use of application data in each system and subsystem shall be identified. The system safety integrity level allocated to each sub-system will determine the standards to be applied to the subsequent development of the data for all installations of the system.

17.4.3.2 During the Software Design phase the detailed interfaces between the generic software and application data shall be specified, unless this has already been specified at an earlier phase of the lifecycle, for example as a result of a requirement to use an existing application specific language.

17.4.3.3 During the Software Module Design phase a rigid separation between program code and data shall be enforced, i.e. it shall be possible to recompile and update either the generic software or the data without needing to update the other, unless there has been a change to the defined interface between the software and data. Likewise, application specific data should be separated from other data.

17.4.3.4 During the Software Maintenance phase, the change control procedures must ensure that any amendment to the generic software may only be installed after it has been established that either the revised software is compatible with the original data, or the data has been revised as necessary.

17.4.3.5 Care must be taken in the Software Verification process and Software Validation phase in order to assure that all relevant combinations of data are considered.

17.4.3.6 The generic software shall be designed to detect corrupted configuration data where this is feasible.

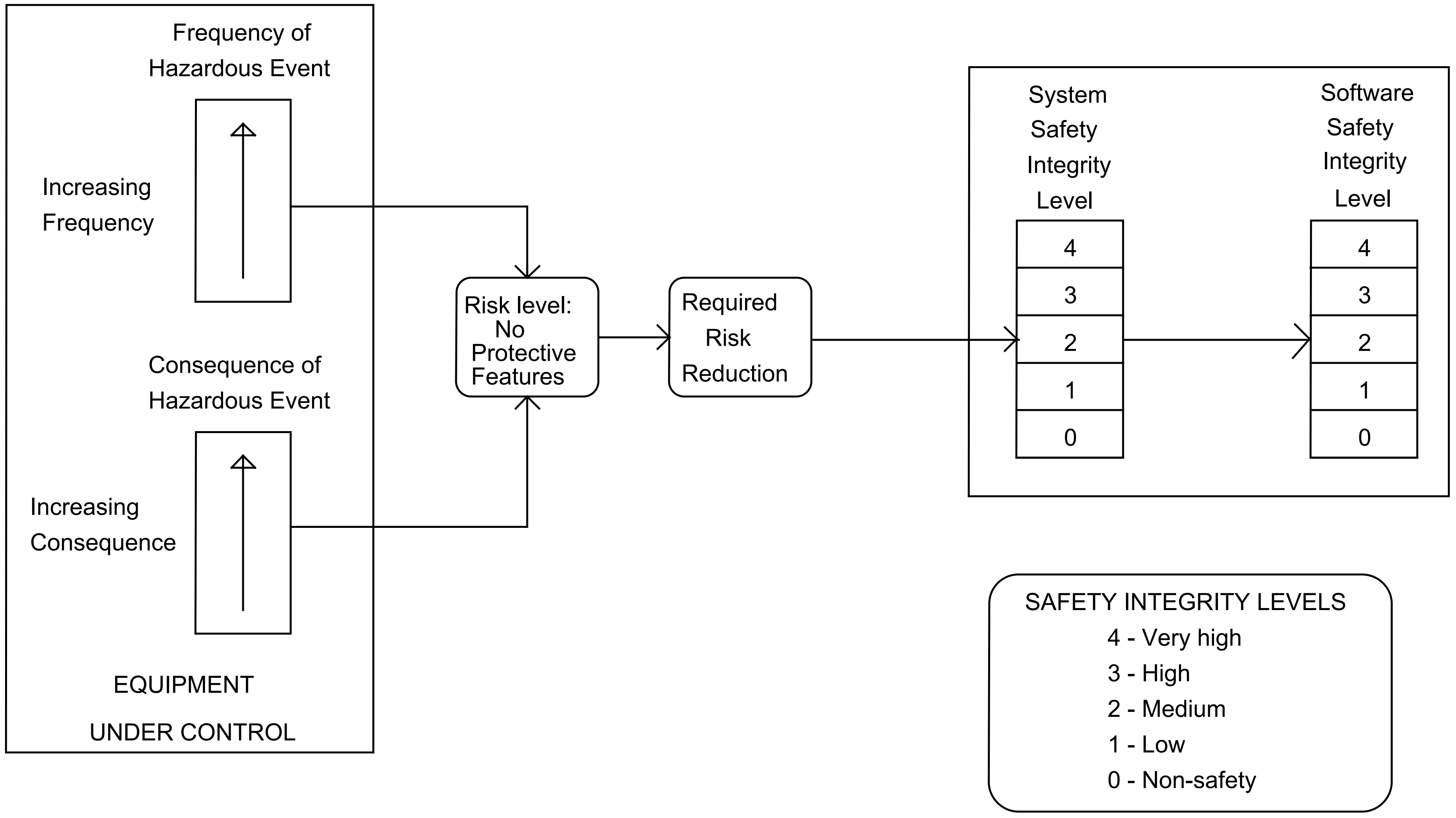


Figure 1 – Integrity Levels for Safety-Related Systems

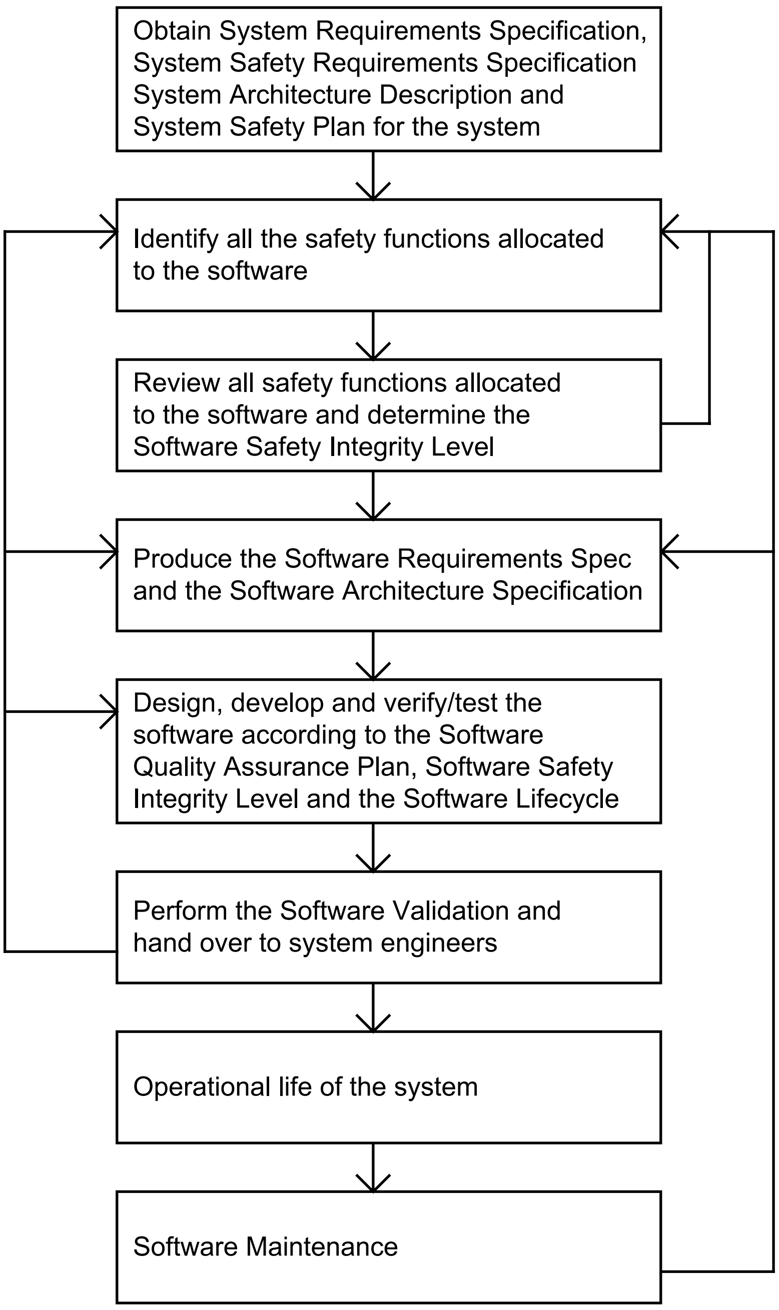

Obtain System Requirements Specification, System Safety Requirements Specification System Architecture Description and System Safety Plan for the system
Identify all the safety functions allocated to the software
Review all safety functions allocated to the software and determine the Software Safety Integrity Level
Produce the Software Requirements Spec and the Software Architecture Specification
Design, develop and verify/test the software according to the Software Quality Assurance Plan, Software Safety Integrity Level and the Software Lifecycle
Perform the Software Validation and hand over to system engineers
Operational life of the system
Software Maintenance

Figure 2 – Software Safety Route Map

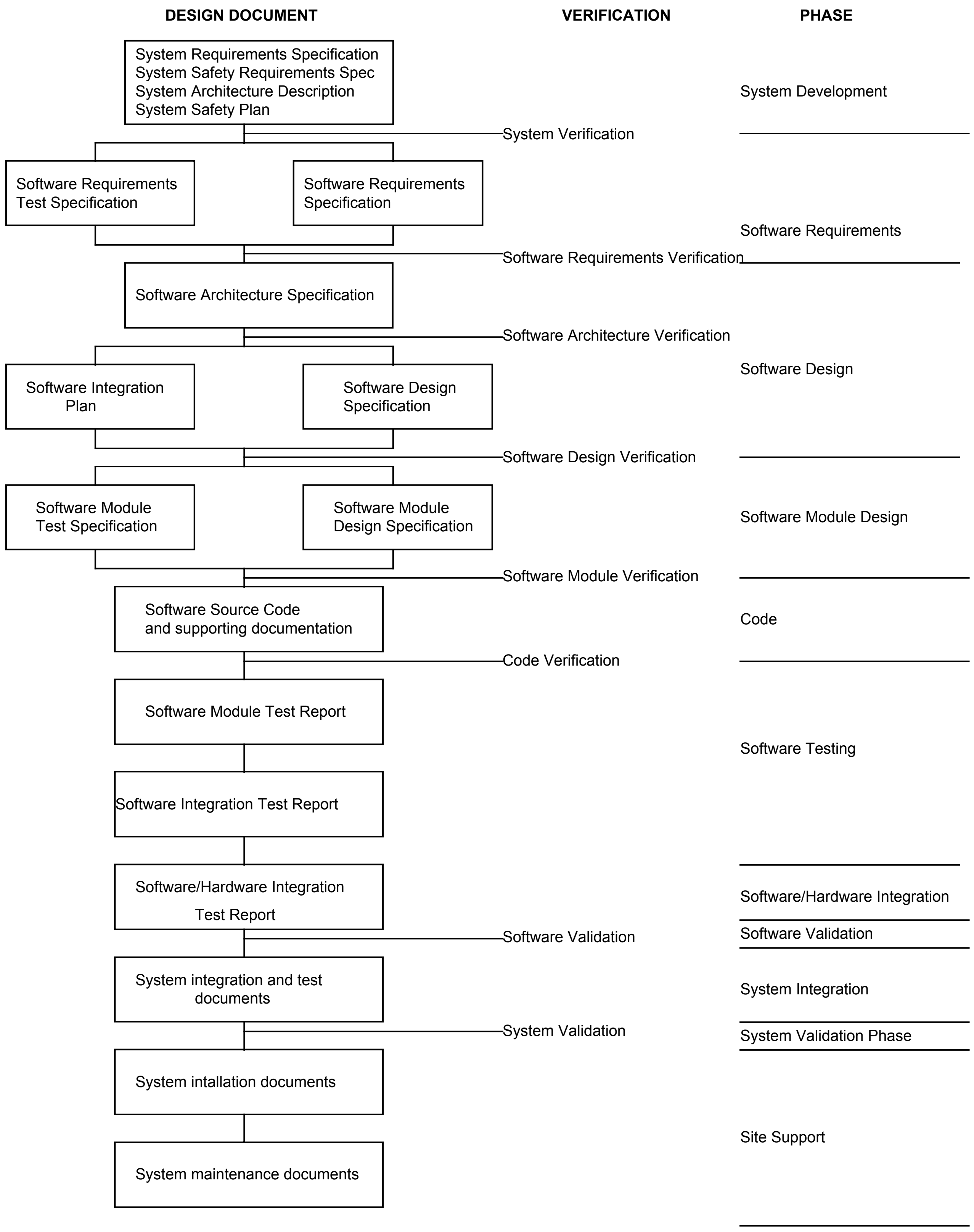
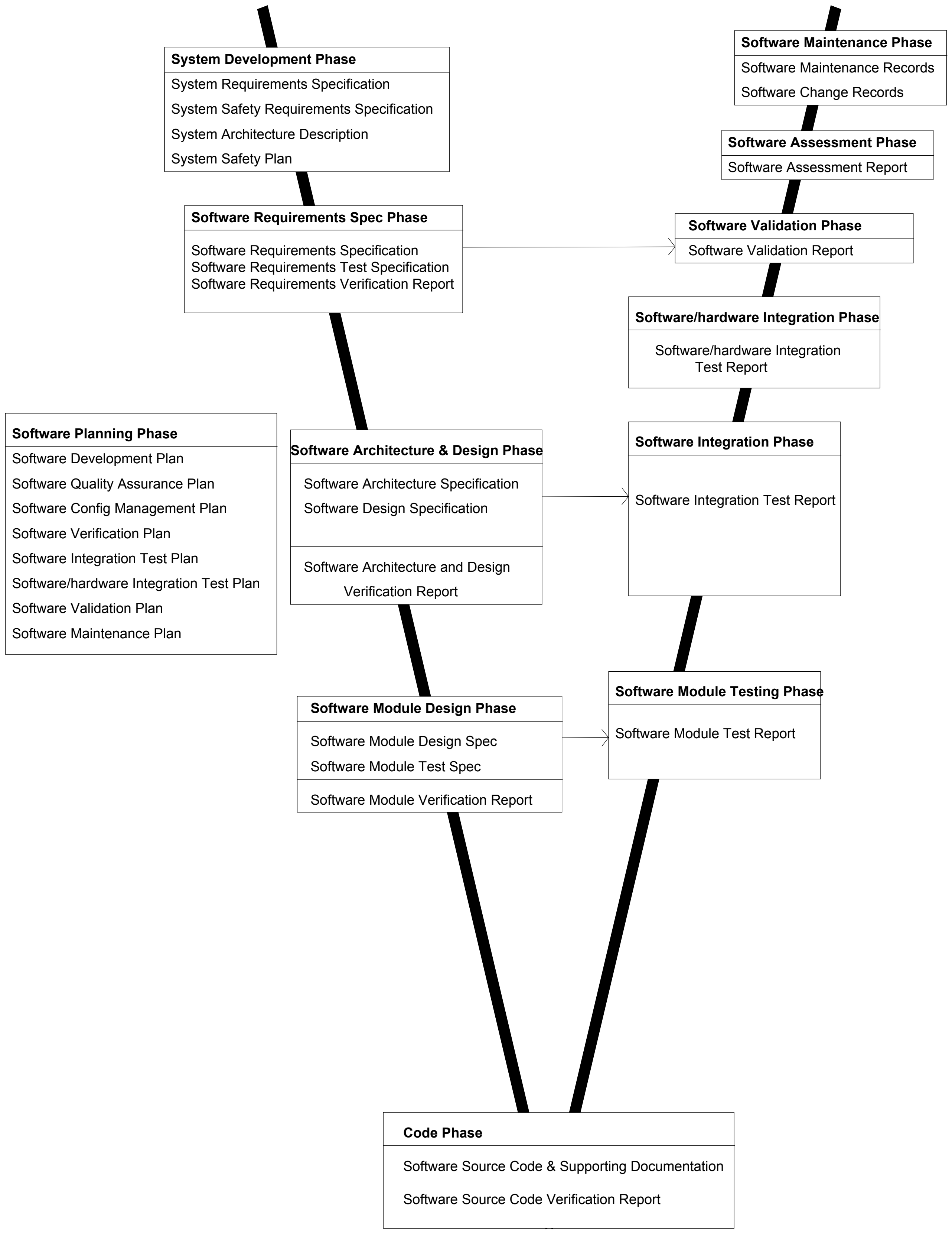

DESIGN DOCUMENT
VERIFICATION
PHASE
System Requirements Specification
System Safety Requirements Spec
System Architecture Description
System Safety Plan
System Development
System Verification
Software Requirements Test Specification
Software Requirements Specification
Software Requirements
Software Requirements Verification
Software Architecture Specification
Software Architecture Verification
Software Design
Software Integration Plan
Software Design Specification
Software Design Verification
Software Module Test Specification
Software Module Design Specification
Software Module Design
Software Module Verification
Software Source Code and supporting documentation
Code
Code Verification
Software Module Test Report
Software Testing
Software Integration Test Report
Software/Hardware Integration Test Report
Software/Hardware Integration
Software Validation
Software Validation
System integration and test documents
System Integration
System Validation
System Validation Phase
System intallation documents
Site Support
System maintenance documents

Figure 3 – Development Lifecycle 1



System Development Phase
System Requirements Specification
System Safety Requirements Specification
System Architecture Description
System Safety Plan
Software Requirements Spec Phase
Software Requirements Specification
Software Requirements Test Specification
Software Requirements Verification Report
Software Planning Phase
Software Development Plan
Software Quality Assurance Plan
Software Config Management Plan
Software Verification Plan
Software Integration Test Plan
Software/hardware Integration Test Plan
Software Validation Plan
Software Maintenance Plan
Software Architecture & Design Phase
Software Architecture Specification
Software Design Specification
Software Architecture and Design Verification Report
Software Module Design Phase
Software Module Design Spec
Software Module Test Spec
Software Module Verification Report
Code Phase
Software Source Code & Supporting Documentation
Software Source Code Verification Report
Software Module Testing Phase
Software Module Test Report
Software Integration Phase
Software Integration Test Report
Software/hardware Integration Phase
Software/hardware Integration Test Report
Software Validation Phase
Software Validation Report
Software Assessment Phase
Software Assessment Report
Software Maintenance Phase
Software Maintenance Records
Software Change Records

Figure 4 – Development Lifecycle 2

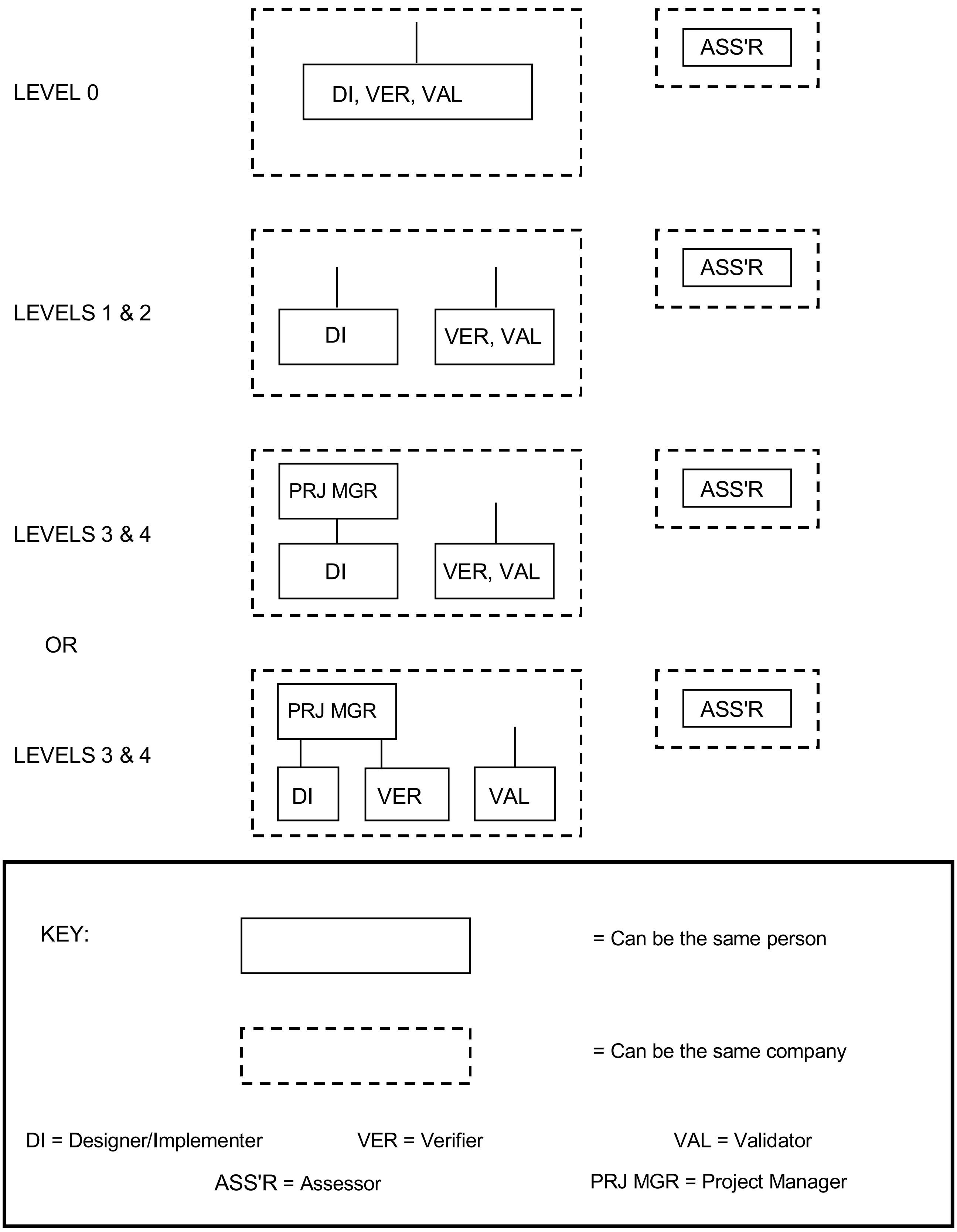


Figure 5 - Independence Versus Software Integrity Level

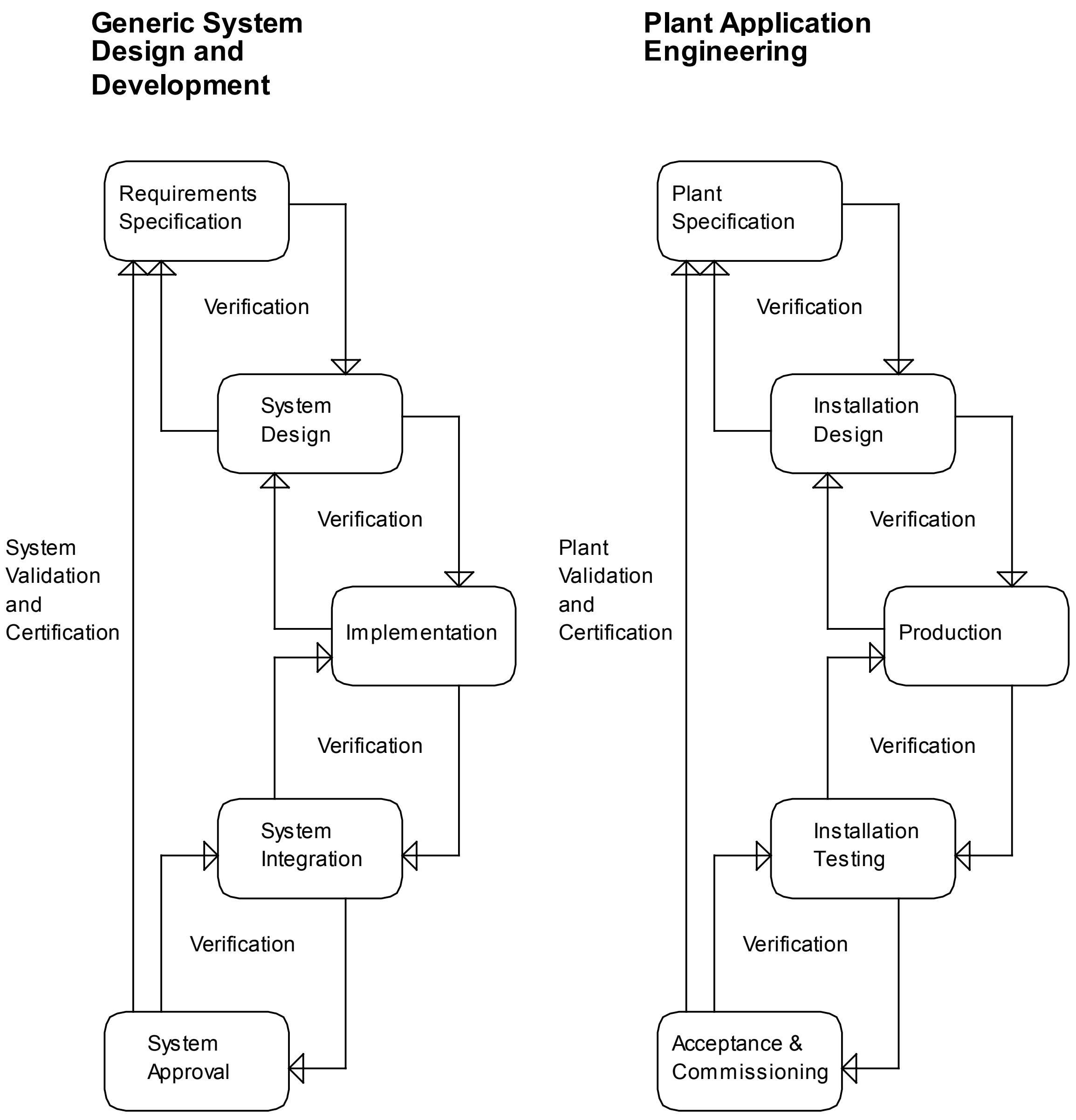


Figure 6 – Relationship between Generic System Development and Application Development

Annex A (normative)

Criteria for the Selection of Techniques and Measures

Each of clauses 7 to 16 of this European Standard has an associated clause table to illustrate the means of achieving conformance. There exist lower level tables, the detailed tables, which expand upon certain entries in the clause tables. For example, Semi Formal Methods in the clause 8 table is expanded upon in the detailed Table D.7. There also exists an informative annex B which is referred to from the clause tables.

With each technique or measure in the tables there is a requirement for each software safety integrity level (SWSIL), 1 to 4 and also for the non safety-related level 0. In this version of the document, the requirements for software safety integrity levels 1 and 2 are the same for each technique. Similarly, each technique has the same requirements at software safety integrity levels 3 and 4. These requirements can be:

'M'	this symbol means that the use of a technique is mandatory;
'HR'	this symbol means that the technique or measure is Highly Recommended for this safety integrity level. If this technique or measure is not used then the rationale behind not using it should be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan;
'R'	this symbol means that the technique or measure is Recommended for this safety integrity level. This is a lower level of recommendation than an 'HR' and such techniques can be combined to form part of a package;
'-'	this symbol means that the technique or measure has no recommendation for or against being used;
'NR'	this symbol means that the technique or measure is positively Not Recommended for this safety integrity level. If this technique or measure is used then the rationale behind using it should be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.

The combination of techniques or measures are to be stated in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan with one or more techniques or measures being selected unless the notes attached to the table makes other requirements. These notes can include reference to approved techniques or approved combinations of techniques. If such techniques or combinations of techniques are used, then the Assessor shall accept them as valid and shall only be concerned that they have been correctly applied. If a different set of techniques is used and can be justified, then the Assessor may find this acceptable.

Clause Tables

Table A.1 – Lifecycle Issues and Documentation (clause 7)

DOCUMENTATION		SWS IL O	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Software Planning Documents	R	HR	HR	HR	HR
2.	S/W Requirements Documents	R	HR	HR	HR	HR
3.	S/W Design Documents	-	HR	HR	HR	HR
4.	S/W Module Documents	-	HR	HR	HR	HR
5.	Source Code & Documentation	R	HR	HR	HR	HR
6.	S/W Test Reports	-	HR	HR	HR	HR
7.	S/W & H/W Integration Test Report	-	HR	HR	HR	HR
8.	S/W Validation Report	R	HR	HR	HR	HR
9.	S/W Assessment Report	-	HR	HR	HR	HR
10.	S/W Maintenance Records	R	HR	HR	HR	HR

Requirement

Compliance with EN ISO 9000-3 implies the production of adequate documentation for all Software Safety Integrity Levels. For Software Safety Integrity Level 0, the designer shall choose suitable types of document.

Table A.2 – Software Requirements Specification (clause 8)

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Formal Methods including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B	B.30	-	R	R	HR	HR
2. Semi-Formal Methods	D.7	R	R	R	HR	HR
3. Structured. Methodology including for example JSD, MASCOT, SADT, SDL, SSADM, and Yourdon.	B.60	R	HR	HR	HR	HR
Requirements 1. The Software Requirements Specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application. 2. The table reflects additional requirements for defining the specification clearly and precisely. One or more of these techniques shall be selected to satisfy the Software Safety Integrity Level being used.						

Table A.3 – Software Architecture (clause 9)

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Defensive Programming	B.15	-	R	R	HR	HR
2.	Fault Detection & Diagnosis	B.27	-	R	R	HR	HR
3.	Error Correcting Codes	B.20	-	-	-	-	-
4.	Error Detecting Codes	B.20	-	R	R	HR	HR
5.	Failure Assertion Programming	B.25	-	R	R	HR	HR
6.	Safety Bag Techniques	B.54	-	R	R	R	R
7.	Diverse Programming	B.17	-	R	R	HR	HR
8.	Recovery Block	B.50	-	R	R	R	R
9.	Backward Recovery	B.5	-	NR	NR	NR	NR
10.	Forward Recovery	B.32	-	NR	NR	NR	NR
11.	Re-try Fault Recovery Mechanisms	B.53	-	R	R	R	R
12.	Memorising Executed Cases	B.39	-	R	R	HR	HR
13.	Artificial Intelligence - Fault Correction	B.1	-	NR	NR	NR	NR
14.	Dynamic Reconfiguration of software	B.18	-	NR	NR	NR	NR
15.	Software Error Effect Analysis	B.26	-	R	R	HR	HR
16.	Fault Tree Analysis	B.28	R	R	R	HR	HR

Requirements

1. Approved combinations of techniques for Software Safety Integrity Levels 3 and 4 shall be as follows:
   a) 1, 7 and one from 4, 5 or 12
   b) 1, 4 and 5
   c) 1, 4 and 12
   d) 1, 2 and 4
   e) 1 and 4, and one of 15 and 16

2. With the exception of entries 3, 9, 10, 13 and 14, one or more of these techniques shall be selected to satisfy the requirements for Software Safety Integrity Levels 1 and 2.

3. Some of these issues may be defined at the system level.

4. Error correcting codes may be used in accordance with the requirements of EN 50159-1 and EN 50159-2.

Table A.4 – Software Design and Implementation (clause 10)

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Formal Methods including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B	B.30	-	R	R	HR	HR
2.	Semi-Formal Methods	D.7	R	HR	HR	HR	HR
3.	Structured. Methodology including for example JSD, MASCOT, SADT, SDL, SSADM and Yourdon.	B.60	R	HR	HR	HR	HR
4.	Modular Approach	D.9	HR	M	M	M	M
5.	Design and Coding Standards	D.1	HR	HR	HR	M	M
6.	Analysable Programs	B.2	HR	HR	HR	HR	HR
7.	Strongly Typed Programming Language	B.57	R	HR	HR	HR	HR
8.	Structured Programming	B.61	R	HR	HR	HR	HR
9.	Programming Language	D.4	R	HR	HR	HR	HR
10.	Language Subset	B.38	-	-	-	HR	HR
11.	Validated Translator	B.7	R	HR	HR	HR	HR
12.	Translator Proven in Use	B.65	HR	HR	HR	HR	HR
13.	Library of Trusted/Verified Modules and Components	B.40	R	R	R	R	R
14.	Functional/ Black-box Testing	D.3	HR	HR	HR	M	M
15.	Performance Testing	D.6	-	HR	HR	HR	HR
16.	Interface Testing	B.37	HR	HR	HR	HR	HR
17.	Data Recording and Analysis	B.13	HR	HR	HR	M	M
18.	Fuzzy Logic	B.67	-	-	-	-	-
19.	Object Oriented Programming	B.68	-	R	R	R	R

Requirements

1. A suitable set of techniques shall be chosen according to the software safety integrity level.

2. At software safety integrity level 3 or 4, the approved set of techniques shall include one of techniques 1, 2 or 3, together with one of techniques 11 or 12. The remaining techniques shall still be treated according to their recommendations.

Table A.5 – Verification and Testing (clause 11)

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Formal Proof	B.31	-	R	R	HR	HR
2.	Probabilistic Testing	B.47	-	R	R	HR	HR
3.	Static Analysis	D.8	-	HR	HR	HR	HR
4.	Dynamic Analysis and Testing	D.2	-	HR	HR	HR	HR
5.	Metrics	B.42	-	R	R	R	R
6.	Traceability Matrix	B.69	-	R	R	HR	HR
7.	Software Error Effect Analysis	B26	-	R	R	HR	HR

Requirements

1. For Software Safety Integrity Level 3 or 4, the approved combinations of techniques shall be:
   a) 1 and 4
   b) 3 and 4
   or c) 4, 6 and 7

2. For Software Safety Integrity Level 1 or 2, the approved technique shall be 1 or 4.

Table A.6 – Software/Hardware Integration (clause 12)

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Functional and Black-box Testing	D.3	HR	HR	HR	HR	HR
2.	Performance Testing	D.6	-	R	R	HR	HR

Requirements

1. For software safety integrity level 0, technique 1 shall be the approved technique.

2. For software safety integrity level 1, 2, 3 or 4, the approved combination of techniques shall be 1 and 2.

Table A.7 – Software Validation (clause 13)

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Probabilistic Testing	B.47	-	R	R	HR	HR
2. Performance Testing	D.6	-	HR	HR	M	M
3. Functional and Black-box Testing	D.3	HR	HR	HR	M	M
4. Modelling	D.5	-	R	R	R	R

Requirement

1. For Software Safety Integrity Level 1, 2, 3 or 4, an approved combination of techniques shall be 2 and 3.

Table A.8 – Clauses to be assessed

CLAUSE TO BE ASSESSED	Clause	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. S/W Safety Integrity Levels	5	HR	HR	HR	HR	HR
2. Personnel & Responsibility	6	-	R	R	HR	HR
3. Lifecycle & Documentation	7	-	HR	HR	HR	HR
4. S/W Requirements Spec.	8	R	HR	HR	HR	HR
5. S/W Architecture	9	-	R	R	HR	HR
6. Design & Development	10	-	R	R	HR	HR
7. Verification	11	-	HR	HR	HR	HR
8. S/W/H/W Integration	12	-	R	R	HR	HR
9. S/W Validation	13	-	HR	HR	HR	HR
10. Quality Assurance	15	-	HR	HR	HR	HR
11. Maintenance	16	-	HR	HR	HR	HR

Table A.9 – Software Assessment (clause 14)
Assessment Techniques

ASSESSMENT TECHNIQUE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Checklists	B.8	HR	HR	HR	HR	HR
2. Static Software Analysis	B.14 B.42 D.8	R	HR	HR	HR	HR
3. Dynamic Software Analysis	D.2 D.3	-	R	R	HR	HR
4. Cause Consequence Diagrams	B.6	R	R	R	R	R
5. Event Tree Analysis	B.23	-	R	R	R	R
6. Fault Tree Analysis	B.28	R	R	R	HR	HR
7. Software Error Effect Analysis	B.26	-	R	R	HR	HR
8. Common Cause Failure Analysis	B.10	-	R	R	HR	HR
9. Markov Models	B.41	-	R	R	R	R
10. Reliability Block Diagram	B.51	-	R	R	R	R
11. Field Trial Before Commissioning	-	R	HR	HR	HR	HR
Requirement 1. One or more of these techniques shall be selected to satisfy the Software Safety Integrity Level being used.						

Table A.10 – Software Quality Assurance (clause 15)

TECHNIQUE/MEASURE	Clause or Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Accredited to EN ISO 9001	15	R	HR	HR	HR	HR
2. Compliant with EN ISO 9000-3	15	M	M	M	M	M
3. Company Quality System	15	M	M	M	M	M
4. Software Configuration Management	B.56	M	M	M	M	M

Table A.11 – Software Maintenance (clause 16)

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Impact Analysis	B.35	R	HR	HR	M	M
2. Data Recording and Analysis	B.13	HR	HR	HR	M	M

Detailed Tables

Table A.12 – Design and Coding Standards (D.1)
Referenced by clause 10

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Coding Standard Exists	B.16	HR	HR	HR	HR	HR
2.	Coding Style Guide	B.16	HR	HR	HR	HR	HR
3.	No Dynamic Objects	B.16	-	R	R	HR	HR
4.	No Dynamic Variables	B.16	-	R	R	HR	HR
5.	Limited Use of Pointers	B.16	-	R	R	R	R
6.	Limited Use of Recursion	B.16	-	R	R	HR	HR
7.	No Unconditional Jumps	B.16	-	HR	HR	HR	HR

Requirement

1. It is accepted that techniques 3, 4 and 5 may be present as part of a validated compiler or translator.
2. A suitable set of techniques shall be chosen according to the software safety integrity level.

Table A.13 – Dynamic Analysis and Testing (D.2)
Referenced by clauses 11 and 14

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Test Case Execution from Boundary Value Analysis	B.4	-	HR	HR	HR	HR
2.	Test Case Execution from Error Guessing	B.21	R	R	R	R	R
3.	Test Case Execution from Error Seeding	B.22	-	R	R	R	R
4.	Performance Modelling	B.45	-	R	R	HR	HR
5.	Equivalence Classes and Input Partition Testing	B.19	-	R	R	HR	HR
6.	Structure-Based Testing	B.58	-	R	R	HR	HR

Requirements

1. The analysis for the test cases is at the sub-system level and is based on the specification and/or the specification and the code.
2. A suitable set of techniques shall be chosen according to the software safety integrity level.

Table A.14 – Functional/Black Box Test (D.3)
Referenced by clauses 10,12, 13 and 14

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Test Case Execution from Cause Consequence Diagrams	B.6	-	-	-	R	R
2.	Prototyping/Animation	B.49	-	-	-	R	R
3.	Boundary Value Analysis	B.4	R	HR	HR	HR	HR
4.	Equivalence Classes and Input Partition Testing	B.19	R	HR	HR	HR	HR
5.	Process Simulation	B.48	R	R	R	R	R

Requirements

1. The completeness of the simulation will depend upon the extent of the software safety integrity level, complexity and application.
2. A suitable set of techniques shall be chosen according to the software safety integrity level.

Table A.15 – Programming Languages (D.4)
Referenced by clause 10

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	ADA	B.62	R	HR	HR	R	R
2.	MODULA-2	B.62	R	HR	HR	R	R
3.	PASCAL	B.62	R	HR	HR	R	R
4.	Fortran 77	B.62	R	R	R	R	R
5.	'C' or C++ (unrestricted)	B.62	R	-	-	NR	NR
6.	Subset of C or C++ with coding standards	B.62 B.38	R	R	R	R	R
7.	PL/M	B.62	R	R	R	NR	NR
8.	BASIC	B.62	R	NR	NR	NR	NR
9.	Assembler	B.62	R	R	R	-	-
10.	Ladder Diagrams	B.62	R	R	R	R	R
11.	Functional Blocks	B.62	R	R	R	R	R
12.	Statement List	B.62	R	R	R	R	R

Requirements

1. At Software Safety Integrity Level 3 and 4 when a subset of languages 1, 2, 3 and 4 are used the recommendation changes to HR.
2. For certain applications the languages 7 and 9 may be the only ones available. At Software Safety Integrity Level 3 and 4 where a Highly recommended option is not available it is strongly recommended that to raise the recommendation to 'R' there should be a subset of these languages and that there should be a precise set of coding standards.
3. For information on assessing the suitability of a programming language see entry in the bibliography for 'Suitable Programming Language', B.62.
4. If a specific language is not in the table, it is not automatically excluded. It should, however, conform to B.62.

Table A.16 – Modelling (D.5)
Referenced by clause 13

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Data Flow Diagrams	B.12	-	R	R	R	R
2. Finite State Machines	B.29	-	HR	HR	HR	HR
3. Formal Methods	B.30	-	R	R	HR	HR
4. Performance Modelling	B.45	-	R	R	HR	HR
5. Time Petri Nets	B.64	-	HR	HR	HR	HR
6. Prototyping/Animation	B.49	-	R	R	R	R
7. Structure Diagrams	B.59	-	R	R	HR	HR
Requirement 1. A suitable set of techniques shall be chosen according to the software safety integrity level.						

Table A.17 –Performance Testing (D.6)
Referenced by clauses 10, 12 and 13

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Avalanche/Stress Testing	B.3	-	R	R	HR	HR
2. Response Timing and Memory Constraints	B.52	-	HR	HR	HR	HR
3. Performance Requirements	B.46	-	HR	HR	HR	HR
Requirement A suitable set of techniques shall be chosen according to the software safety integrity level.						

Table A.18 – Semi-Formal Methods (D.7)
Referenced by clauses 8 and 10

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Logic/Function Block Diags	-	R	R	R	HR	HR
2. Sequence Diagrams	-	R	R	R	HR	HR
3. Data flow Diagrams	B.12	R	R	R	R	R
4. Finite State Machines/State Transition Diagrams	B.29	-	R	R	HR	HR
5. Time Petri Nets	B.64	-	R	R	HR	HR
6. Decision/Truth Tables	B.14	R	R	R	HR	HR
Requirement A suitable set of techniques shall be chosen according to the software safety integrity level.						

Table A.19 – Static Analysis (D.8)
Referenced by clauses 11 and 14

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Boundary Value Analysis	B.4	-	R	R	HR	HR
2. Checklists	B.8	-	R	R	R	R
3. Control Flow Analysis	B.9	-	HR	HR	HR	HR
4. Data Flow Analysis	B.11	-	HR	HR	HR	HR
5. Error Guessing	B.21	-	R	R	R	R
6. Fagan Inspections	B.24	-	R	R	HR	HR
7. Sneak Circuit Analysis	B.55	-	-	-	R	R
8. Symbolic Execution	B.63	-	R	R	HR	HR
9. Walkthroughs/Design Reviews	B.66	HR	HR	HR	HR	HR
Requirement A suitable set of techniques shall be chosen according to the software safety integrity level.						

Table A.20 – Modular Approach (D.9)
Referenced by clause 10

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Module Size Limited	B.43	HR	HR	HR	HR	HR
2.	Information Hiding/Encapsulation	B.36	R	HR	HR	HR	HR
3.	Parameter Number Limit	B.43	R	R	R	R	R
4.	One Entry/One Exit Point in Subroutines and Functions	B.43	R	HR	HR	HR	HR
5.	Fully Defined Interface	B.43	HR	HR	HR	M	M
Requirement A suitable set of techniques shall be chosen according to the software safety integrity level.							

Annex B (informative)

Bibliography of techniques

B.1 AI Fault Correction (Referenced by clause 9)

Aim

To be able to react to possible hazards in a very flexible way by introducing a mix (combination) of methods and process models and some kind of on-line safety and reliability analysis.

Description

In particular fault forecasting (calculating trends), fault correction, maintenance and supervisory actions may be supported by AI-based systems in a very efficient way in diverse channels of a system, since the rules might be derived directly from the specifications and checked against these. Certain common faults which are introduced into specifications by implicitly already having some design and implementation rules in mind may be avoided effectively by this approach, especially when applying a combination of models and methods in a functional or descriptive manner.

The methods are selected such that faults may be corrected and the effects of failures be minimised, in order to meet the desired safety and reliability.

B.2 Analysable Programs (Referenced by clause 10)

Aim

To design a program in a way that program analysis is easily feasible. The program behaviour must be testable completely on the basis of the analysis.

Description

The intention is to produce programs which are easy to analysis using static analysis methods. In order to achieve this, the rules of structured programming should be followed, for instance:

- the module control flow should be composed of structured constructs, that is sequences, iterations and selection.
- the modules should be small.
- the number of possible paths through a module is small.
- the individual program parts have to be designed so that they are decoupled as far as possible.
- the relation between the input and output parameters should be as simple as possible.
- complex calculations should not be used as the basis of branching and loop decisions.
- branch and loop decisions should be simply related to the module input parameters.
- boundaries between different types of mappings shall be simple.

References:

Verification - The Practical Problems. B.J.T. Webb and D.J. Mannering, SARSS 87, Nov. 1987, Altrincham, England, Elsevier Applied Science, ISBN 1-85166-167-0, 1987.

An Experience in Design and Validation of Software for a Reactor Protection System.S. Bologna, E. de Agostino et. al., IFAC Workshop, SAFECOMP 1979, Stuttgart, 16-18 May1979, Pergamon Press 1979.

B.3 Avalanche/Stress Testing (Referenced by D.6)

Aim

To burden the test object with an exceptionally high workload in order to show that the test object would stand normal workloads easily.

Description

There are a variety of test conditions which can be applied for avalanche/stress testing. Some of these test conditions are listed below:

– if working in a polling mode then the test object gets much more input changes per time unit as under normal conditions;

– if working on demands then the number of demands per time unit to the test object is increased beyond normal conditions;

– if the size of a database plays an important role then it is increased beyond normal conditions;

– influential devices are tuned to their maximum speed or lowest speed respectively;

– for the extreme cases, all influential factors, as far as is possible, are put to the boundary conditions at the same time.

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

B.4 Boundary Value Analysis (Referenced by D.2, D.3 and D.8)

Aim

To remove software errors occurring at parameter limits or boundaries.

Description

The input domain of the program is divided into a number of input classes. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention:

– zero divisor;

– blank ASCII characters;

– empty stack or list element;

– null matrix;

– zero table entry.

Normally the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also, if it is possible to specify a test case which causes output to exceed the specification boundary values.

If output is a sequence of data, for example a printed table, special attention should be paid to the first and the last elements and to lists containing none, 1 and 2 elements.

References:

The Art of Software Testing. G. Myers, Wiley & Sons, New York, USA, 1979.

B.5 Backward Recovery (Referenced by clause 9)

Aim

To provide correct functional operation in the presence of one or more faults.

Description

If a fault has been detected, the system is reset to an earlier internal state, the consistency of which has been proven before. This method implies saving of the internal state frequently at so-called well defined checkpoints. This may be done globally (for the complete database) or incremental (changes only between checkpoints). Then the system has to compensate for the changes which have taken place in the meantime by using journalling (audit trail of actions), compensation (all effects of these changes are nullified) or external (manual)interaction.

B.6 Cause Consequence Diagrams (Referenced by clause 14 and D.3)

Aim

To model, in a diagrammatic form, the sequence of events that can develop in a system as a consequence of combinations of basic events.

Description

It can be regarded as a combination of fault-tree and event-tree analysis. Starting from a critical event, a cause-consequence graph is traced backwards and forwards. In the backwards direction it is equivalent to a fault tree with the critical event as the given top event. In the forward direction the possible consequences arising from an event are identified. The graph can contain vertex symbols which describe the conditions for propagation along different branches from the vertex. Time delays can also be included. These conditions can also be described with fault trees. The lines of propagation can be combined with logical symbols, to make the diagram more compact. A set of standard symbols are defined for use in cause consequence diagrams. The diagrams can be used to compute the probability of occurrence of certain critical consequences.

References:

The Cause Consequence Diagram Method as a Basis for Quantitative Accident Analysis.D.S. Nielsen, Riso-M-1374, 1971.

B.7 Certified Tools and Certified Translators (Referenced by clause 10)

Aim

Tools are necessary to help developers in the different phases of software development. Wherever possible tools should be certified so that some level of confidence can be assumed regarding the correctness of the outputs.

Description

A certified tool is one which has been determined to be of a particular quality. The certification of a tool will generally be carried out by an independent, often national, body, against independently set criteria, typically national or international standards. Ideally, the tools used in all development phases (definition, design, coding, testing and validation) and those used in configuration management, should be subject to certification. To date only compilers (translators) are regularly subject to certification procedures; these are laid down by national certification bodies and they exercise compilers (translators) against international standards. such as those for Ada and Pascal.

References:

Pascal Validation Suite. UK distributor: BSI Quality Assurance, PO Box 375, Milton Keynes, MK14 6LL.

Ada Validation Suite. UK distributor: National Computing Centre (NCC), Oxford Road., Manchester England.

B.8 Checklists (Referenced by clause 14 and D.8)

Aim

To provide a stimulus to critical appraisal of all aspects of the system rather than to lay down specific requirements.

Description

A set of questions to be completed by the person performing the checklist. Many of the questions are of a general nature and the Assessor must interpret them as seems most appropriate to the particular system being assessed.

To accommodate wide variations in software and systems being validated, most checklists contain questions which are applicable to many types of system. As a result there may well be questions in the checklist being used which are not relevant to the system being dealt with and which should be ignored. Equally there may be a need, for a particular system, to supplement the standard checklist with questions specifically directed at the system being dealt with.

In any case it should be clear that the use of checklists depends critically on the expertise and judgement of the engineer selecting and applying the checklist. As a result the decisions taken by the engineer, with regard to the checklist(s) selected, and any additional or superfluous questions, should be fully documented and justified. The objective is to ensure that the application of the checklists can be reviewed and that the same results will be achieved unless different criteria are used.

The object in completing a checklist is to be as concise as possible. When extensive justification is necessary this should be done by reference to additional documentation. Pass, Fail and Inconclusive, or some similar restricted set of tokens should be used to record the results for each question. This conciseness greatly simplifies the process of reaching an overall conclusion as to the results of the checklist assessment.

References:

Programmable Electronic Systems in Safety Related Applications. Health and Safety Executive, Her Majesty's Stationary Office, London 1987.

Guidelines for the Assessment of the Safety and Reliability of High Integrity Industrial Computer Systems. EWICS TC7, WP 489, 6th October 1987.

The Art of Software Testing. G. Myers, Wiley & Sons, New York, USA 1979.

Software for computers in the safety systems of nuclear power stations. IEC 60880, 1986.

B.9 Control Flow Analysis (Referenced by D.8)

Aim

To detect poor and potentially incorrect program structures.

Description

Control Flow Analysis identifies suspect areas of code which do not follow good programming practice. The program is analysed to form a directed graph which can be analysed for:

– inaccessible code, for instance, unconditional jumps which leaves blocks of code unreachable;

– knotted code, which is well structured code whose control graph is reducible by successive graph reductions to a single node. Poorly structured code can only be reduced to a knot composed of several nodes.

References:

RXVP80 - The Verification and Validation System for FORTRAN. Users Manual. General Research Corporation, Santa Barbara, California, USA.

Information Flow and Data Flow of While Programs. J.F. Bergeretti and B.A. Carre, ACMTrans. on Prog. Lang. and Syst., 1985.

B.10 Common Cause Failure Analysis (Referenced by clause 14)

Aim

To identify potential failures in redundant systems or redundant sub-systems which would undermine the benefits of redundancy because of the appearance of the same failures in the redundant parts at the same time.

Description

Computer systems intended to take care of the safety of a plant often use redundancy in hardware and majority voting. This technique is used to avoid random component failures, which would tend to prevent the correct processing of data in a computer system.

However, some failures can be common to more than one component. For example, if a computer system is installed in one single room, shortcomings in the air-conditioning, might reduce the benefits of redundancy. The same is true for other external effects on the computer system such as: fire, flooding, electromagnetic interference, plane crashes, and earthquakes. The computer system may also be affected by incidents related to operation and maintenance. It is essential, therefore, that adequate and well documented procedures are provided for operation and maintenance. Extensive training of operating and maintenance personnel is also essential.

Internal effects are also major contributors to Common-Cause Failures (CCF). They can stem from design errors in common or identical components and their interfaces, as well as ageing of components. CCF-Analysis has to search the system for such potential common failures. Methods of CCF-Analysis are general quality control, design reviews, verification and testing by an independent team, and analysis of real incidents with feedback of experience from similar systems. The scope of the analysis, however, goes beyond hardware. Even if 'diverse software' is used in difficult chains of a redundant computer system, there might be some commonality in the software approaches which could give rise to CCF. Errors in the common specification, for example.

When CCF's do not occur exactly at the same tine, precautions can be taken by means of comparison methods between the redundant chains which should lead to detection of a failure before this failure is common to all chains. CCF analysis should take this technique into account.

References:

Review of Common Cause Failures. I.A. Watson, UKAEA, Centre for Systems Reliability, Wigshaw Lane, WA3 4NE, England, NCSR R 27, July 1981.

Common-Mode Failures in Redundancy Systems. I.A. Watson and G.T. Edwards Nuclear Technology Vol, 46, De. 1979.

Programmable Electronic Systems in Safety Related Applications. Health and Safety Executive, Her Majesty's Stationary Office.

B.11 Data Flow Analysis (Referenced by D.8)

Aim

To detect poor and potentially incorrect program structures.

Description

Data Flow Analysis combines the information obtained from the control flow analysis with information about which variables are read or written in different portions of code. The analysis can check for:

– variables that are read before they are written. This is very likely to be an error, and is certainly bad programming practice;

– variables that are written more than once without being read. This could indicate omitted code;

– variables that are written but never read. This could indicate redundant code.

There is an extension of data flow analysis known as information flow analysis, where the actual data flows (both within and between procedures) are compared with the design intent. This is normally implemented with a computerised tool where the intended data flows are defined using a structured comment that can be read by the tool.

References:

RXVP80 - The Verification and Validation System for FORTRAN. Users Manual. General Research Corporation, Santa Barbara, California, USA.

Information Flow and Data Flow of While Programs. J.F. Bergeretti and B.A. Carre, ACMTrans. on Prog. Lang. and Syst., 1985.

B.12 Data Flow Diagrams (Referenced by D.5 and D.7)

Aim

To describe the data flow through a program in a diagrammatic form.

Description

Data Flow Diagrams document how data input is transformed to output, with each stage in the diagram representing a distinct transformation.

Data flow diagrams are made up of three components:

1. annotated arrows - represent data flow in and out of the transformation centres with the annotations specifying what the data is;

2. annotated bubbles - represent transformation centres with the annotation specifying the transformation;

3. operators (AND, XOR) - These operators are used to link the annotated arrows.

Data flow diagrams describe how an input is transformed to an output. They do not, and should not, include control information or sequencing information. Each bubble can be considered as a stand alone black box which, as soon as its inputs are available, transforms them to its outputs.

One of the principle advantages of data flow diagrams is that they show transformations without making any assumptions about how these transformations are implemented.

The preparation of data flow diagrams is best approached by considering system inputs and working towards system outputs. Each bubble must represent a distinct transformation - its output should, in some way, be different from its input. There are no rules for determining the overall structure of the diagram and constructing a data flow diagram is one of the creative aspects of system design. Like all design, it is an iterative process with early attempts refined in stages to produce the final diagram.

References:

Software Engineering. I. Sommerville, Addison-Wesley 1982. ISBN 0-201-13795-X.

B.13 Data Recording and Analysis (Referenced by clauses 10 and 16)

Aim

To record all data, decisions and rationale in the software project to allow for easier verification, validation, assessment and maintenance.

Description

Detailed records are maintained during a project, both on a project and individual basis. For instance, an engineer would be required to keep records which could include

– effort expanded on individual modules,

– testing performed on each module,

– decisions and their rationale,

– achievement of project milestones,

– problems and their solutions.

During and at the conclusion of the project these records can be analysed to establish a wide variety of information. In particular data recording is very important for the maintenance of computer systems as the rationale for certain decisions made during the development project is not always known by the maintenance engineers.

B.14 Decision Tables (Truth Tables) (Referenced by clause 14 and D.7)

Aim

To provide a clear and coherent specification and analysis of complex logical combinations and relationships.

Description

These related methods use two dimensional tables to concisely describe logical relationships between Boolean program variables.

The conciseness and tabular nature of both methods makes them appropriate as a means of analysing complex logical combinations expressed in code.

Both methods are potentially executable if used as specifications.

B.15 Defensive Programming (Referenced by clause 9)

Aim

To produce programs which detect anomalous control flow, data flow or data values during their execution and react to these in a predetermined and acceptable manner.

Description

Many techniques can be used during programming to check for control or data anomalies. These can be applied systematically throughout the programming of a system to decrease the likelihood of erroneous data processing.

Two overlapping areas of defensive techniques can be identified. Intrinsic error-safe software is designed to accommodate its own design shortcomings. These shortcomings may be due to plain error of design or coding, or to erroneous requirements. The following lists some of the defensive techniques:

– variables should be range checked;

– where possible, values should be checked for plausibility;

– parameters to procedures should be type, dimension and range checked at procedure entry.

These first three recommendations help to ensure that the numbers manipulated by the program are reasonable, both in terms of the program function and physical significance of the variables.

– Read-only and read-write parameters should be separated and their access checked. Functions should treat all parameters as read-only. Literal constants should not be write-accessible. This helps detect accidental overwriting or mistaken use of variables.

Error tolerant software is designed to 'expect' failures in its own environment or use outside nominal or expected conditions, and behave in a predefined manner. Techniques include the following:

– input variables and intermediate variables with physical significance should be checked for plausibility;

– the effect of output variables should be checked, preferably by direct observation of associated system state changes;

– the software should check its configuration. This could include both the existence and accessibility of expected hardware and also that the software itself is complete. This is particularly important for maintaining integrity after maintenance procedures.

Some of the defensive programming techniques such as control flow sequence checking, also cope with external failures.

References:

Dependability of Critical Computer Systems - Part 1. E.F. Redmill (ed) Elsevier Applied Science 1988.

Dependability of Critical Computer systems - Part 2. E.F. Redmill (ed) Elsevier Applied Science 1988.

Software Engineering Aspects of Real-time Programming Concepts. E. Schoitsch, Computer Physics Communications 41 (1986) North Holland, Amsterdam.

B.16 Design and Coding Standards (Referenced by D.1)

Aim

To ensure a uniform layout of the design documents and the produced code, enforce egoless programming and to enforce a standard design method.

Description

The rules to be adhered to are agreed at the outset of the project between the participants. These rules must comprise as a minimum

– the development method and the related coding standards to be followed, for example JSP, MASCOT, Petri-Nets etc;

– details of modularisation, for example, interface shapes, module sizes;

- use of encapsulation, inheritance and polymorphy, in case of object oriented languages;
- use or avoidance of certain language constructs, for example GOTO, EQUIVALENCE, dynamic objects, dynamic data, recursion, pointers, exits etc.; and
- the what, where and how to comment.

These rules are made to allow for ease of development, verification, assessment and maintenance. Therefore they shall address the available tools, in particular analysers and reverse engineering tools.

B.17 Diverse Programming (Referenced by clause 9)

Aim

Detect and mask residual software design faults during execution of a program, in order to prevent safety critical failures of the system, and to continue operation for high reliability.

Description

In diverse programming a given program specification implemented N times in different ways. The same input values are given to the N versions, and the results produced by the N versions are compared. If the result is considered to be valid, the result is transmitted to the computer outputs.

The N versions can run in parallel on separate computers, alternatively all versions can be run on the same computer and the results subjected to an internal vote. Different voting strategies can be used on the N versions depending on the application requirements.

- If the system has a safe state, then it is feasible to demand complete agreement (all N agree) otherwise a fail-safe output value is used. For simple trip systems the vote can be biased in the safe direction. In this case the safe action would be to trip if either version demanded a trip. This approach typically uses only two versions (N=2).
- For systems with no safe state, majority voting strategies can be employed. For cases where there is no collective agreement, probabilistic approaches can be used in order to maximise the chance of selecting the correct value, for example, taking the middle value, temporary freezing of outputs until agreement returns etc.

This technique does not eliminate residual software design faults, but it provides a measure to detect and mask before they can affect safety.

References:

Dependable Computing : From Concepts to Design Diversity. A. Avizienis, J.C. Laprie, Proc IEEE, Vol 74, 5, May 1986.

A Theoretical Basis for the Analysis of Multi-version Software subject to Co-incident Failures. D.E. Eckhardt and L.D. Lee, IEEE Trans. SE-11, No 12, 1985.

B.18 Dynamic Reconfiguration (Referenced by clause 9)

Aim

To maintain system functionality despite an internal fault.

Description

The logical architecture of the system has to be such that it can be mapped onto a subset of the available resources of the system. The architecture needs to be capable of detecting a failure in a physical resource and then remapping the logical architecture back onto the restricted resources left functioning. Although the concept is more traditionally restricted to recovery from failed hardware units, it is also applicable to failed software units if there is sufficient 'run-time redundancy' to allow a software re-try or if there is sufficient redundant data to make the individual and isolated failure a little import.

Although traditionally applied to hardware, this technique is being developed for application to software and, thus, the total system. It must be considered at the first system design stage.

References:

Critical Issues in the Design of a Reconfigureable Control Computer. H. Schmid, J. Lam,R. Naro and K. Weir, FTCS 14 June 1984 IEEE 1984.

Assigning Processes to Processors : A Fault-tolerant Approach. June 1984 G. Kar and C.N.Nikolaou, Watson Research Centre, Yorktown

B.19 Equivalence Classes and Input Partition Testing (Referenced by D2 and D.3)

Aim

To test the software adequately using a minimum of test data. The test data is obtained by selecting the partitions of the input domain required to exercise the software.

Description

This testing strategy is based on the equivalence relation of the inputs, which determines a partition of the input domain.

Test cases are selected with the aim of covering all subsets of this partition. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning which are:

– equivalence classes may be defined on the specification. The interpretation of the specification may be either input oriented, for example the values selected are treated in the same way or output oriented, for example the set of values leading to the same functional result; and

– equivalence classes may be defined on the internal structure of the program. In this case the equivalence class results are determined from static analysis of the program, for example the set of values leading to the same path being executed.

References:

The Art of Software Testing. G. Myers, Wiley & Sons, New York, USA, 1979.

B.20 Error Detecting and Correcting Codes (Referenced by clause 9)

Aim

To detect and correct errors in sensitive information.

Description

For an information of n bits, a coded block of k bits is generated which enables errors to be detected and corrected. Different types of code include:

– hamming codes;

– cyclic codes;

– polynomial codes.

References:

The Technology of Error Correcting Codes. E.R. Berlekamp, Proc. of the IEEE, 68, 5, 1980.

A Short Course on Error Correcting Codes. N.J.A. Sloane, Springer-Verlag, Wien, 1975.

B.21 Error Guessing (Referenced by D.2 and D.8)

Aim

To remove common programming errors.

Description

Testing experience and intuition combined with knowledge and curiosity about the system under test may add some uncategorised test cases to the designed test case set. Special values or combinations of values may be error-prone. Some interesting test cases may be derived from inspection checklists. It may also be considered whether the system is robust enough. Can the buttons be pushed on the front-panel too fast or too often? What happens if two buttons are pushed simultaneously?

B.22 Error Seeding (Referenced by D.2)

Aim

To ascertain whether a set of test cases is adequate.

Description

Some known error types are inserted in the program, and the program is executed with the test cases under test conditions. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is an estimate of the ratio of found real errors to total number errors. This gives a possibility of estimating the number of remaining errors and thereby the remaining test effort.

$$\frac{\text{Found seeded errors}}{\begin{array}{c}\text{Total number of}\\ \text{seeded errors}\end{array}} = \frac{\text{Found real errors}}{\begin{array}{c}\text{Total number of}\\ \text{real errors}\end{array}}$$

The detection of all the seeded errors may indicate either that the test case set is adequate, or that the seeded errors were too easy to find. The limitations of the method are that in order, to obtain any usable results, the error types as well as the seeding positions must reflect the statistical distribution of real errors.

B.23 Event Tree Analysis (Referenced by clause 14)

Aim

To model, in a diagrammatic form, the sequence of events that can develop in a system after an initiating event, and thereby indicate how serious consequences can occur.

Description

On the top of the diagram is written the sequence conditions that are relevant in the development following the initiating event which is the target of the analysis. Starting under the initiating event, one draws a line to the first condition in the sequence. There the diagram branches off into a 'yes' and a 'no' branch, describing how the future developments depend on the condition. For each of these branches, one continues to the next condition in a similar way. Not all conditions are, however, relevant for all branches. One continues to the end of the sequence, and each branch of the tree constructed in this way represents a possible consequence. The event tree can be used to compute the probability of the various consequences based on the probability and number of conditions in the sequence.

References:

Event Trees and their Treatment on PC Computers. N. Limnious and J.P. Jeannette, Reliability Engineering, Vol. 18, No. 3 1987.

B.24 Fagan Inspections (Referenced by D.8)

Aim

To reveal errors in all phases of the program development.

Description

A 'formal' audit on quality assurance documents aimed at finding errors and omissions. The inspection process consists of five phases; Planning, Preparation, Inspection, Rework and Follow up. Each of these phases has its own separate objective. The complete system development (specification, design, coding and testing) must be inspected.

References:

Design and Code Inspections to Reduce Errors in Program Development. M.E. Fagan, IBM Systems Journal, No. 3, 1976.

B.25 Failure Assertion Programming (Referenced by clause 9)

Aim

To detect residual software design faults during execution of a program, in order to prevent safety critical failures of the system and to continue operation for high reliability.

Description

The assertion programming method follows the idea of checking a pre-condition(before a sequence of statements is executed, the initial conditions are checked for validity) and a post-condition (results are checked after the execution of a sequence of statements). If either the pre-condition or the post-condition is not fulfilled, the processing stops with an error.

For example,

```
assert <pre-condition>;
    action 1;
        :
        :
    action x;
assert <post-condition>;
```

References:

A Discipline of Programming. E.W. Dijkstra, Prentice Hall, 1976.

The Science of Programming. D. Gries, Springer-Verlag, 1981.

Software Development - A Rigorous Approach. C.B. Jones, Prentice-Hall, 1980.

B.26 SEEA - Software Error Effect Analysis (Ref'd by clause 9, 11 & 14)

Aim

To identify software modules, their criticality; to propose means for detecting software errors and enhancing software robustness; to evaluate the amount of validation needed on the various software components.

Description

The analysis is done in three phases:

- Vital software modules identification;

  Determination of the depth of the analysis (at the level of a single instruction line, a group of instructions, a module, etc...) needed for each software module, from its specification.

- Software error analysis

  The result of this phase is a table listing the following information:

  - module name;
  - error considered;
  - consequences of the error at the module level;
  - consequences at the system level;
  - violated safety criterion;
  - error criticality;
  - proposed error detection means;
  - violated criterion if the detection means is implemented;
  - residual criticality if the detection means is implemented.

- Synthesis

  The synthesis identifies the remaining unsafe scenarios and the validation effort needed given the criticality of each module.

SEEA, being an in-depth analysis carried out by an independent team, is a powerful bug-finding method.

References:

Railway fixed equipment and rolling stock. Data processing. Software dependability - Adapted methods for software safety analysis, French standard NF F 71-013, December 1990.

IRSE International Technical Committee Report No. 1
SAFETY SYSTEM VALIDATION with regard to cross acceptance of signalling systems by the railways

P. THIREAU
Méthodologie d'Analyse des Effets des Erreurs Logiciel (AEEL) appliquée à l'étude d'un logiciel de haute sécurité.
5th International Conference on Reliability and Maintainability.
Biarritz - France - 1986

D. J. REIFER
Software failures modes and effects analysis
Transaction on reliability IEE - Vol. R28 No. 3
August 79 - Pages 247/249

J. R. FRAGOLA and J. F. SPAMM
The software error effects analysis, a qualitative design tool
IEEE Symposium Computer on Software Reliability
1973 - pages 90/93

B.27 Fault Detection and Diagnosis (Referenced by clause 9)

Aim

To detect faults in a system, which might lead to a failure, thus providing the basis for countermeasures in order to minimise the consequences of failures.

Description

Fault detection is the process of checking a system for erroneous states (which are caused, as explained before, by a fault within the (sub)system to be checked). The primary goal of fault detection is to inhibit the effect of wrong results. A system which delivers either correct results, or no results at all, is called "self checking".

Fault detection is based on the principles of redundancy (mainly to detect hardware faults) and diversity (software faults). Some sort of voting is needed to decide on the correctness of results. Special methods applicable are assertion programming, N-version programming and the safety bag technique and on hardware level by introducing sensors, control loops, error checking codes, etc.

Fault detection may be achieved by checks in the value domain or in the time domain on different levels, especially on the physical (temperature, voltage etc.), logical (error detecting codes), functional (assertions) or external level (plausibility checks). The results of these checks may be stored and associated with the data affected to allow failure tracking.

Complex systems are composed of subsystems. The efficiency of fault detection, diagnosis and fault compensation depends on the complexity of the interactions among the subsystems, which influences the propagation of faults.

Fault diagnosis isolates the smallest subsystem that may be identified. Smaller subsystems allow a more detailed diagnosis of faults (identification of erroneous states).

B.28 Fault Tree Analysis (Referenced by clauses 9 & 14)

Aim

To aid in the analysis of events, or combinations of events, that will lead to a hazard or serious consequence.

Description

Starting at an event which would be the immediate causes of a hazard or serious consequence (the 'top event') analysis is carried out along a tree path. Combinations of causes are described with logical operators (and, or, etc). Intermediate causes are analysed in the same way, and so on back to basic events where analysis stops.

The method is graphical, and a set of standardised symbols are used to draw the fault tree. It is mainly intended for the analysis of hardware systems, but there have also been attempts to apply this approach to software failure analysis.

References:

System Reliability Engineering Methodology: A Discussion of the State of the Art. J.B. Fusseland J.S. Arend, Nuclear Safety 20(5), 1979.

Fault Tree Handbook. W.E. Vesely et. al., NUREG-0942, Division of System Safety Office of Nuclear Reactor Regulation, U.S. Nuclear Regulatory Commission Washington, D.C.20555, 1981.

Reliability Technology. A.E. Greene and A.J. Bourne, Wiley-Interscience, 1972.

B.29 Finite State Machines/State Transition Diagrams (Ref'd by D.5 & D.7)

Aim

To define or implement the control structure of a system.

Description

Many systems can be defined in terms of their states, their inputs, and their actions. Thus when in state S1, on receiving input I a system might carry out action A and move to state S2. By defining a system's actions for every input in every state we can define a system completely. The resulting model of the system is called a Finite State Machine. It is often drawn as a so-called state transition diagram showing how the system moves from one state to another, or as a matrix in which the dimensions are state and input and the matrix cells contain the action and new state resulting from receipt of the input in the given state.

Where a system is complicated or has a natural structure this can be reflected in a layered FSM.

A specification or design expressed as an FSM can be checked for completeness (the system must have an action and new state for every input in every state), for consistency (only one state change is defined for each state/input pair) and reachability (whether or not it is possible to get from one state to another by any sequence of inputs). These are important properties for critical systems and they can be checked. Tools to support these checks are easily written. Algorithms also exist that allow the automatic generation of test cases for verifying an FSM implementation or for animating an FSM model.

References:

Introduction to the theory of Finite State Machines. A. Gill, McGraw-Hill 1962.

B.30 Formal Methods (Referenced by clause 8, clause 10 and D.5)

Aim

The development of software in a way that is based on mathematics. This includes formal design and formal coding techniques.

Description

Formal methods provide a means of developing a description of a system at some stage in its development specification, design or code. The resulting description takes a mathematical form and can be subjected to mathematical analysis to detect various classes of inconsistency or incorrectness. Moreover, the description can in some cases be analysed by machine with a rigour similar to the syntax checking of a source program by a compiler, or animated to display various aspects of the behaviour of the system described. Animation can give extra confidence that the system meets the real requirement as well as the formally specified requirement.

A formal method will generally offer a notation (generally some form of discrete mathematics being used), a technique for deriving a description in that notation, and various forms of analysis for checking a description for different correctness properties.

Several examples of Formal Methods are described in the following subsections of this bibliography. The Formal Methods described are CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM and Z.

B.30.1 CCS - Calculus of Communicating Systems

Aim

CCS is a means for describing and reasoning about the behaviour of systems of concurrent, communicating processes.

Description

Similar to CSP, CCS is a mathematical calculus concerned with the behaviour of systems. The system design is modelled as a network of independent processes operating sequentially or in parallel. Processes can communicate via ports (similar to CSP's channels),the communication only taking place when both processes are ready. Non-determinism can be modelled. Starting from a high-level abstract description of the entire system (known as a trace), it is possible to carry out a step-wise refinement of the system into a composition of communicating processes whose total behaviour is that required of the whole system. Equally, it is possible to work in a bottom up fashion, combining processes and deducing the properties of the resulting system using inference rules related to the composition rules.

References:

A Calculus of Communicating Systems. R Milner, Report number ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK.

The Specification of Complex Systems. B Cohen, W T Harwood, M I Jackson. Addison-Wesley, 1986.

B.30.2 CSP - Communicating Sequential Processes

Aim

CSP is a technique for the specification of concurrent software systems, i.e. systems of communicating processes operating concurrently.

Description

CSP provides a language for the specification of systems of processes and proof for verifying that the implementation of processes satisfies their specifications (described as a trace - permissible sequences of events).

A system is modelled as a network of independent processes. Each process is described in terms of all of its possible behaviours. A system is modelled by composing processes sequentially or in parallel. Processes can communicate (synchronise or exchange data) via channels, the communication only taking place when both processes are ready. The relative timing of events can be modelled.

The theory behind CSP was directly incorporated into the architecture of the INMOS transputer, and the OCCAM language allows a CSP-specified system to be directly implemented on a network of transputers.

References:

Communicating Sequential Processes. C A R Hoare, Prentice-Hall, 1985

B.30.3 HOL - Higher Order Logic

Aim

This is a formal language intended as a basis for hardware specification and verification.

Description

HOL (Higher Order Logic) refers to a particular logic notation and its machine support system, both of which were developed at the University of Cambridge Computer Laboratory. The logic notation is mostly taken from Church's Simple Theory of Types and the machine support system is based upon the LCF (Logic of Computable Functions) system.

References:

HOL : A Machine Orientated Formulation of Higher Order Logic. M. Gordon, University of Cambridge Technical Report, Number - 68.

B.30.4 LOTOS

Aim

LOTOS is a means for describing and reasoning about the behaviour of systems of concurrent, communicating processes.

Description

LOTOS (Language for Temporal Ordering Specification) is based on CCS with additional features from the related algebras CSP and CIRCAL (Circuit Calculus). It overcomes the weakness of CCS in the handling of data structures and value expressions by combining it with a second component based on the abstract data type language ACT ONE. The process definition component of LOTOS could, however, be used with other formalisms for the description of abstract data types.

Reference:

Information Processing Systems - Open Systems Inter-connection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO DIS 8807, July 20, 1987.

B.30.5 OBJ

Aim

To provide a precise system specification with user feed-back and system validation prior to implementation.

Description

OBJ is an algebraic specification language. Users specify requirements in terms of algebraic equation. The behavioural, or constructive, aspects of the system are specified in terms of operations acting on abstract data types (ADT). An ADT is like an ADA package where the operator behaviour is visible whilst the implementation details are 'hidden'.

An OBJ specification, and subsequent step-wise implementation, is amenable to the same formal proof techniques as other formal approaches. Moreover, since the constructive aspects of the OBJ specification are machine-executable, it is straightforward to achieve system validation from the specification itself. Execution is essentially the evaluation of a function by equation substitution (re-writing) which continues until specific output value is obtained. This executability allows end-users of the envisaged system to gain a 'view' of the eventual system at the system specification stage without the need to be familiar with the underlying formal specification techniques.

As with all other ADT techniques, OBJ is only applicable to sequential systems, or to sequential aspects of concurrent systems. OBJ has been widely used for the specification of both small and large-scale industrial applications.

References:

An introduction to OBJ; A language for Writing and Testing Specifications. J A Goguen and JTardo, Specification of Reliable Software, IEEE Press 1979, reprinted in Software Specification Techniques, N Gehani, A McGettrick (eds), Addison-Wesley, 1985.

Algebraic Specification for Practical Software Production. C Rattray, Cogan Press, 1987.

An Algebraic Approach to the Standardisation and Certification of Graphics Software. R Gnatz, Computer Graphics Forum 2(2/3), 1983.

DTI STARTS Guide. 1987, NCC, Oxford Road, Manchester, UK.

B.30.6 Temporal Logic

Aim

Direct expression of safety and operational requirements and formal demonstration that these properties are preserved in the subsequent development steps.

Description

Standard First Order Predicate Logic contains no concept of time. Temporal logic extends First Order logic by adding modal operators (e.g. 'Henceforth' and 'Eventually'). These operators can be used to qualify assertions about the system. For example, safety properties might be required to hold 'henceforth', whilst other desired system states might be required to be attained 'eventually' from some other initiating state. Temporal formulas are interpreted on sequences of states (behaviours). What constitutes a 'state' depends on the chosen level of description. It can refer to the whole system, a system component or the computer program. Quantified time intervals and constraints are not handled explicitly in temporal logic. Absolute timing has to be handled by creating additional time states as part of the state definition.

References:

Temporal Logic of Programs. F Kroger EATCS Monographs on Computer Science, Vol 8, Springer Verlag, 1987.

Design for Safety using Temporal Logic. J Gorski. SAFECOMP 86, Sarlat France, Pergamon Press, October 1986.

Logics for Computer Programming. D Gabay. Ellis Horwood.

B.30.7 VDM - Vienna Development Method

Aim

The systematic specification and implementation of sequential programs.

Description

VDM is a mathematically based specification technique and a technique for refining implementations in a way that allows proof of their correctness with respect to the specification.

The specification technique is model-based in that the system state is modelled in terms of set-theoretic structures on which are defined invariants (predicates), and operations on that state are modelled by specifying their pre and post-conditions in terms of the system state. Operations can be proved to preserve the system invariants.

The implementation of the specification is done by the reification of the system state in terms of data structures in the target language and by refinement of the operations in terms of a program in the target language. Reification and refinement steps give rise to proof obligations that establish their correctness. Whether or not these obligations are carried out is a choice made by the designer.

VDM is principally used in the specification stage but can be used in the design and implementation stages leading to source code. It can only be applied to sequential programs or the sequential processes in concurrent systems.

References:

Software Development - A Rigorous Approach. C B Jones. Prentice-Hall, 1980.

Formal Specification and Software Development. D Bjorner & C B Jones. Prentice-Hall, 1982.

Systematic Software Development using VDM. C B Jones. Prentice-Hall, 1986.

The Specification of Complex Systems. B Cohen, W T Harwood and M I Jackson. Addison-Wesley, 1986.

B.30.8 Z and B

Aim

Z is a specification language notation for sequential systems and a design technique that allows the developer to proceed from a Z specification to executable algorithms in a way that allows proof of their correctness with respect to the specification.

Z is principally used in the specification stage but a method has been devised to go from specification into a design and an implementation. It is best suited to the development of data oriented, sequential systems.

B is an associated method.

Description

Like VDM, the specification technique is model-based in that the system state is modelled in terms of set-theoretic structures on which are defined invariants (predicates), and operations on that state are modelled by specifying their pre and post-conditions in terms of the system state. Operations can be proved to preserve the system invariants thereby demonstrating their consistency. The formal part of a specification is divided into schemas which allow the structuring of specifications through refinement.

Typically, a Z specification is a mixture of formal Z and informal explanatory text in natural language. (Formal text on its own can be too terse for easy reading and often its purpose needs to be explained, while the informal natural language can easily become vague and imprecise).

Unlike VDM, Z is a notation rather than a complete method. However an associated method (called B) has been developed which can be used in conjunction with Z. The B method is based on the principle of step-wise refinement.

References:

The Z Notation - A Reference Manual. J M Spivey, Prentice Hall, 1988.

Specification Case Studies. Edited by I Hayes, Prentice-Hall, 1987.

Specification of the UNIX Filestore. C Morgan and B Sufrin. IEEE Transactions on Software Engineering, SE-10, 2 March 1984.

The B Book - Assigning Programs to Meanings. J R Abrial, Cambridge United Press, 1996.

B.31 Formal Proof (Referenced by clause 11)

Aim

Using theoretical and mathematical models and rules it is possible to prove the correctness of a program without executing it.

Description

A number of assertions are stated at various locations in the program, and they are used as pre and post conditions to various paths in the program. The proof consists of showing that the program transfers the preconditions into the post-conditions according to a set of logical rules, and that the program terminates.

Several Formal Methods are described in this bibliography, for instance, CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM and Z. The descriptions of these can be found under the section 'Formal Methods'.

References:

Can Program Proving be made Practical. J. Dahl, Research Report, ISBN 82-90230-26-5 No33, Oslo, May

B.32 Forward Recovery (Referenced by clause 9)

Aim

To provide correct functional operation in the presence of one or more faults.

Description

If a fault has been detected, the current state of the system is manipulated to obtain a state, which will be consistent some time later. This concept is especially suited for real-time systems with a small database and fast rate of change of the internal state. It is assumed, that at least part of the system state may be imposed onto the environment, and only part of the system states are influenced (forced) by the environment.

B.33 Graceful Degradation

Aim

To maintain the more critical system functions available despite failures by dropping the less critical functions.

Description

This technique gives priorities to the various functions to be carried out by the system. The design then ensures that should there be insufficient resources to carry out all the system functions, then the higher priority functions are carried out in preference to the lower ones. For example, error and event logging functions may be lower priority than system control functions. System control would continue if the hardware associated with error logging were to fail. Further, should the system control hardware fail, but not the error logging hardware, then the error logging hardware would take over the control function. This is predominantly applied to hardware but is applicable to the total system. It must betaken into account from the top-most design phase.

References:

Space Shuttle Software. C.T. Sheridan, Datamation, Vol 24, July 1978

The Evolution of Fault-Tolerant Computing. Vol 1 of Dependable Computing and Fault Tolerant Systems, Edited by A. Avizienis, H. Kopetz, and J.C. Laprie, Springer-Verlag, ISBN 3-211-81941-X, 1987.

Fault Tolerance, Principle and Practices. Vol 3 of [2] above, T. Anderson and P.A. Lee, Springer-Verlag, ISBN 3-211-82077-9, 1987.

B.34 Hazard and Operability Study (HAZOP)

Aim

To establish, by a series of systematic examinations of the component sections of the computer system and its operation, failure modes which lead to result in potentially hazardous situations in the controlled system.

Typical hazardous events on the controlled systems are fire, explosion, release of toxic material (chemical or nuclear) or serious economic loss.

It is assumed that the hazardous events on the system being controlled have been identified in a separate Hazard Analysis and the consequence of the hazardous events classified into degrees of seriousness.

The analysis which covers all stages of the project life-cycle, from specification through design to maintenance and modification, and is intended to identify at each stage events/failure modes which could lead to potential hazards and thus eliminate them.

Description

The analysis is carried out by a team of engineers (covering computer, instrument, electrical, process, safety and operational disciplines) led by a trained specialist in hazard analysis techniques in a series of scheduled meetings.

It is important that a fixed time schedule is allocated within the project for the meetings; each one scheduled for at least half a day and for effectiveness no more than four per week to allow for maintaining the flow of accompanying documentation.

Prior to the study, agreed checklists for the systematic examination will have been compiled and for each section of the system leading questions such as; What if it happens? How can it happen? When can it happen? Does it matter? are asked. When positive answers are given further questions are asked, such as; What can be done about it? When must it be done? Is there an alternative? etc.

For the first part of the analysis it is suggested that the computer installation as a whole is examined. The second and subsequent parts involve detailed examination of the relevant parts of the computer systems itself.

At each part or stage of the analysis, the objective is to identify failure modes which lead to potential hazards on the controlled system and the degree of their effect. The major component parts of the computer system are further sub-divided and where necessary subjected to a separate analysis.

At suitable points throughout the systematic analysis, action review meetings will be arranged.

It is essential that comprehensive records of the meetings are kept, for they will form a substantial part of the system hazard/safety dossier.

After a number of meetings it is suggested that a review meeting be held to ensure that actions are followed up, modifications suggested during the Study meetings are incorporated into the study etc.

References:

HAZOP and HAZAN. T.A. Kletz, 1986, 2nd Edition, Institution of Chemical Engineers, 165-171 Railway Terrace, Rugby, CV1 3HQ, UK.

Reliability and Hazard Criteria for Programmable Electronic Systems in the Chemical Industry. E. Johnson, Proc of 'Safety and Reliability of PES', PES 3 Safety Symposium, B.K. Daniels (ed), 28-30 May 1986, Guernsey Channel Islands, Elsevier Applied Science, 1986.

Reliability Engineering and Risk Assessment. E.J. Henlty and H. Kumamoto, Prentice Hall,1981.

Systems Reliability and Risk Analysis. E.G. Frenkel, Martinus Nijhogg 1984.

B.35 Impact Analysis (Referenced by clause 16)

Aim

To identify the effect that a change or an enhancement to a software system will have to other modules in that software system as well as to other systems.

Description

Prior to a modification or enhancement being performed on the software an analysis shall be undertaken to identify the impact of the modification or enhancement on the software and to also identify the affected software systems and modules.

After the analysis has been completed a decision is required concerning the reverification of the software system. This depends on the number of modules affected, the criticality of the affected modules and the nature of the change. The possible decisions are:

i) only the changed module to be reverified;

ii) all identified affected modules are reverified; and

iii) the complete system is reverified.

B.36 Information Hiding / Encapsulation (Referenced by D.9)

Aim

To increase the reliability and maintainability of software.

Description

Data that is globally accessible to all software components can be accidentally or incorrectly modified by any of these components. Any changes to these data structures may require detailed examination of the code and extensive modifications.

Information hiding is a general approach for minimising these difficulties. The key data structures are 'hidden' and can only be manipulated through a defined set of access procedures. This allows the internal structures to be modified or further procedures to be added without affecting the functional behaviour of the remaining software. For example, a name directory might have access procedures Insert, Delete and Find. The access procedures and internal data structures could be re-written (e.g. to use a different look-up method or to store the names on a hard disk) without affecting the logical behaviour of the remaining software using these procedures.

This concept of an abstract data type is directly supported in a number of programming languages, but the basic principle can be applied whatever programming language is used.

References:

Software Engineering: Planning for Change, D A Lamb, Prentice Hall, 1988.

On the Design and Development of Program Families, D L Parnas, IEEE Trans SE-2, Mar 1976.

B.37 Interface Testing (Referenced by clause 10)

Aim

To demonstrate that interfaces of subprograms do not contain any errors or any errors that lead to failures in a particular application of the software or to detect all errors that may be relevant.

Description

Several levels of detail or completeness of testing are feasible. The most important levels are testing:

– all interface variables at their extreme positions;

– all interface variable individually at their extreme values with other interface variables at normal values;

– all values of the domain of each interface variable with other interface variables at normal values;

– all values of all variables in combination. this may only be feasible for small interfaces;

– the specified test conditions relevant to each call of each subroutine.

These tests are particularly important if their interfaces do not contain assertions that detect incorrect parameter values. They are also important after new configurations of pre-existing subprograms have been generated.

B.38 Language Subset (Referenced by clause 10 and D.4)

Aim

To reduce the probability of introducing programming faults and increase the probability of detecting any remaining faults.

Description

The language is examined to identify programming constructs which are either error-prone or difficult to analyse, for example, using static analysis methods. A language subset is then defined which excludes these constructs.

B.39 Memorising Executed Cases (Referenced by clause 9)

Aim

To force the software to fail safe if it executes an unlicensed path.

Description

During licensing a record is made of all relevant details of each program execution. During normal operation each program execution is compared with the set of the licensed executions. If it differs, a safety action is taken.

The execution record can be the sequence of the individual decision-to-decision paths (DDpaths) or the sequence of the individual accesses to arrays, records or volumes, or both.

Different methods of storing execution paths are possible. Nash-coding methods can be used to map the execution sequence onto a single large number or sequence of numbers. During normal operation the execution path value must be checked against the stored cases before any output operation occurs.

Since the possible combinations of decision-to-decision paths during one program is very large, it may not be feasible to treat programs as a whole. In this case, the technique can be applied at module level.

Reference:

Fail-safe Software - Some Principles and a Case Study. W. Ehrenberger, Proc. SARSS 1987,Altringham, Manchester, UK, Elsevier Applied Science, 1987.

B.40 Library of Trusted/Verified Modules and Components (Ref'd by clause 10)

Aim

To avoid the need for software modules and hardware component designs to be extensively revalidated or redesigned for each new application. Also to advantage designs which have not been formally or rigorously validated but for which considerable operational history is available.

Description

Well designed and structured PESs are made up of a number of hardware and software components and modules which are clearly distinct and which interact with each other in clearly defined ways.

Different PESs designed for differing applications will contain a number of modules or components which are the same or very similar. Building up a library of such generally applicable modules allows much of the resource necessary for validating the designs to be shared by more than one application.

Furthermore the use of such modules in multiple applications provides empirical evidence of successful operational use. This empirical evidence justifiably enhances the trust which users are likely to have in the modules.

B.41 Markov Models (Referenced by clause 14)

Aim

To evaluate the reliability, safety or availability of a system.

Description

A graph of the system is constructed. The graph represents the status of the system with regard to its failure states (the failure states are represented by the nodes of the graph). The edges between nodes, which represent the failure events or repair events, are weighted with the corresponding failure rates or repair rates. Note that the failure events, states and rates can be detailed in such a way that a precise description of the system is obtained, e.g. detected or undetected failures, manifestation of a larger failure etc.

The Markov technique is suitable for modelling redundant systems in which the level of redundancy varies with time due to component failure and repair. Other classical methods, for example, FMEA and FTA, cannot readily be adapted to modelling the effects of failures throughout the life-cycle of the system since no simple combinatorial formulae exist for calculating the corresponding probabilities.

In the simplest cases, the formulae which describe the probabilities of the system are readily available in the literature or can be calculated manually. In more complex cases, some methods of simplification (absorbing states) exist. For very complex cases results can be calculated by computer simulation of the graph.

References:

The Theory of Stochastic Processes. R.E. Cox and H.D. Miller, Methuen and Co. Ltd,London, UK, 1968

Finite MARKOV Chains. J.G. Kemeny and J.L. Snell, D. Van Nostrand Company Inc.,Princeton, 1959

Reliability Handbook. B.A. Koslov and L.A. Ushakov, Holt Rinehart and Winston Inc.,New York 1970

The Theory and Practice of Reliable System Design. D.P. Siewiorek and R.S. Swarz, Digital Press 1982.

B.42 Metrics (Referenced by clause 11 and clause 14)

Aim

To predict the attributes of programs from properties of the software itself rather than from its development or test history.

Description

These models evaluate some structural properties of the software and relate this to a desired attribute such as reliability or complexity. Software tools are required to evaluate most of the measures. Some of the metrics which can be applied are given below:

- Graph Theoretic Complexity: this measure can be applied early in the lifecycle to assess trade-offs, and is based on the complexity of the program control graph, represented by its cyclomatic number;
- number of ways to activate a certain module (accessibility): the more a module can be accessed, the more likely it is to be debugged;
- Software Science: this measure computes the program length by counting the number of operators and operands. It provides a measure of complexity and estimates development resources;
- number of entries and exits per module: minimising the number of entry/exit points is a key feature of structured design and programming techniques.

References:

A Complexity Measure. T.J. McCabe, IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, Dec 1976.

Models and Measurements for Quality Assessments of Software. S.N. Mohanty, ACMComputing Surveys, Vol 11, No. 3, Sep 1979.

Elements of Software Science. M.H. Halstead, Elsevier, North Holland, New York, 1977.

B.43 Modular Approach (Referenced by D.9)

Aim

Decomposition of a software systems into small comprehensible parts in order to limit the complexity of the system.

Description

A Modular Approach or modularisation contains several rules for the design, coding and maintenance phases of a software project. These rules vary according to the design method employed during design. Most methods contain the following rules:

- a module shall have a single well defined task or function to fulfil;
- connections between modules shall be limited and strictly defined, coherence in one module shall be strong;
- collections of subprograms shall be built providing several levels of modules;
- subprogram sizes shall be restricted to some specified value typically 2 to 4 screen sizes;
- subprograms shall have a single entry and a single exit only;
- modules shall communicate with other modules via their interfaces. Where global or common variables are used they shall be well structured, access shall be controlled and their use shall be justified in each instance;
- all module interfaces shall be fully documented;

– each module shall hide something from its environment;

– any modules interface shall contain the minimum number of parameters necessary for the modules function; and

– a suitable restriction of parameter number shall be specified, typically 5.

B.44 Monte-Carlo Simulation

Aim

To simulate phenomenon of the real world in the software using random numbers.

Description

Monte-Carlo simulations are used to solve problems. These problems fall into two classes:

– probabilistic, where random numbers are used to generate a real world stochastic phenomenon; and

– deterministic, which are mathematically translated into an equivalent probabilistic problem.

Monte-Carlo simulation injects a random number streams to simulate noise on an analytic signal or to add random biases or tolerances. The Monte-Carlo simulation is run to produce a large sample from which statistical results are obtained.

When using Monte-Carlo simulations care must be taken to ensure that the biases, tolerances or noise are reasonable values.

A general principle of Monte-Carlo simulations is to restate and reformulate the problem so that the results obtained are as accurate as possible rather than tackling the problem as initially stated.

B.45 Performance Modelling (Referenced by D.2 and D.5)

Aim

To ensure that the working capacity of the system is sufficient to meet the specified requirements.

Description

The requirements specification includes throughput and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. The proposed system design is compared against the stated requirements by

– defining a model of the system processes, and their interactions,

– identifying the use of resources by each process, for example, processor time, communications bandwidth, storage devices etc),

– identifying the distribution of demands placed upon the system under average and worst-case conditions,

– computing the mean and worst-case throughput and response times for the individual system functions.

For simple systems, an analytic solution may be possible whilst for more complex systems, some form of simulation is required to obtain accurate results.

Before detailed modelling, a simpler 'resource budget' check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is infeasible. Even if the design passes this check, performance modelling may show that excessive delays and response times occur due to resource starvation. To avoid this situation engineers often design systems to use some fraction (e.g. 50 %) of the total resources so that the probability of resource starvation is reduced.

B.46 Performance Requirements (Referenced by D.6)

Aim

To establish that the performance requirements of a software system have been satisfied.

Description

An analysis is performed of both the system and the software requirements specifications to identify all general and specific, explicit and implicit performance requirements.

Each performance requirement is examined in turn to determine

– the success criteria to be obtained,

– whether a measure against the success criteria can be obtained,

– the potential accuracy of such measurements,

– the project stages at which the measurements can be estimated, and

– the project stages at which the measurements can be made.

The practicability of each performance requirement is then analysed in order to obtain a list of performance requirements, success criteria and potential measurements. The main objectives are;

i) each performance requirement is associated with at least one measurement;

ii) where possible, accurate and efficient measurements are selected which can be used as early in the development process as possible;

iii) essential and optional performance requirements and success criteria are identified and

iv) where possible, advantage shall be taken of the possibility of using a single measurement for more than one performance requirement.

B.47 Probabilistic Testing (Referenced by clause 11 and clause 13)

Aim

To gain a quantitative figure about the reliability properties of the investigated software. This figure may address the related levels of confidence and significance and

i) a failure probability per demand,

ii) a failure probability during a certain period of time, and

iii) a probability of error containment.

From these figures other parameters may be derived such as

- probability of failure free execution,
- probability of survival,
- availability,
- MTBF or failure rate, and
- probability of safe execution.

Description

Probabilistic considerations are either based on a probabilistic test or on operating experience. Usually the number of tests cases of observed operating cases is very large.

In order to facilitate testing, usually automatic aids are taken. They concern the details of test data provision and test output supervision. Large tests are run on large host computers with the appropriate process simulation periphery. Test data is selected both according to systematic and random view points. The first concern the overall test control, for example, guarantee a test data profile. The random selection takes the individual test cases in detail.

Individual test harnesses, test executions and test supervisions are determined by the detailed test aims as described above. Other important conditions are given through the mathematical prerequisites to be fulfilled in order to enable the test evaluation in view of the intended test aim.

Probabilistic figures about the behaviour of any test object may also be derived from operating experience. Provided the same conditions are met, the same mathematics can be applied as for the evaluation of test results.

B.48 Process Simulation (Referenced by D.3)

Aim

To test the function of a software system, together with its interface to the outside world, without allowing it to modify the real world in any way.

Description

The creation of a system, for testing purposes only, which mimics the behaviour of the system to be controlled by the system under test.

The simulation may be software only or a combination of software and hardware. It must

- provide all the inputs of the system under test which will exist when the system is installed,
- respond to outputs from the system in a way which faithfully represents the controlled plant,
- have provision for operator inputs to provide any perturbations with which the system under test is required to cope.

When software is being tested the simulation may be a simulation of the target hardware with its inputs and outputs.

References:

A Software Simulator - An Aid to Plant Commissioning. S. Nunns, EWICS TC7.

Physical Fault Simulation. F. Morillon, EWICS Document number WP460.

B.49 Prototyping / Animation (Referenced by D.3 and D.5)

Aim

To check the feasibility of implementing the system against the given constraints. To communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings.

Description

A sub-set of system functions, constraints, and performance requirements are selected. A prototype is built using high level tools. At this stage, constraints such as the target computer, implementation language, program size, maintainability, reliability and availability need not be considered. The prototype is evaluated against the customer's criteria and the system requirements may be modified in the light of this evaluation.

References:

Proc. Working Conference on Prototyping. Namur Oct 1983, Budde et al, Springer-verlag, 1984.

Using an executable specification language for an information system. S. Urban et. al, IEEE Trans Software Engineering, Vol. SE-11 No. 7 July 1985.

B.50 Recovery Block (Referenced by clause 9)

Aim

To increase the likelihood of the program performing its intended function.

Description

Several different program sections are written, often independently, each of which is intended to perform the same desired function. The final program is constructed from these sections. The first section, called the primary, is executed first. This is followed by an acceptance test of the result it calculates. If the test is passed then the result is accepted and passed on to subsequent parts of the system. If it fails, any side effects of the first are reset and the second section, called the first alternative, is executed. This too is followed by an acceptance test and is treated as in the first case. A second, third or even more alternatives can be provided if desired.

Reference:

System Structure for Software Fault Tolerance. B. Randall, IEEE Trans Software Engineering, Vol SE-1, No 2, 1975.

B.51 Reliability Block Diagram (Referenced by clause 14)

Aim

To model, in a diagrammatic form, the set of events that must take place and conditions which must be fulfilled for a successful operation of a system or a task.

Description

The target of the analysis is represented as a success path consisting of blocks, lines and logical junctions. A success path starts from one side of the diagram and continues via the blocks and junctions to the other side of the diagram. A block represents a condition or an event, and the path can pass it if the condition is true or the event has taken place. If the path comes to a junction it continues if the logic of the junction is fulfilled. If it reaches a vertex, it may continue along all outgoing lines. If there exists at least one success path through the diagram the target of the analysis is operating correctly.

References:

System Reliability Engineering Methodology : A Division of the State of the Art. J.B. Fusseland J.S. Arend, Nuclear Safety 20(5), 1979.

Fault Tree Handbook. W.E. Vesely et al, NUREG-0942, Division of System Safety Office at Nuclear Reactor Regulation, U.S. Nuclear Regulatory Commission, Washington, D.C. 20555, 1981.

B.52 Response Timing and Memory Constraints (Referenced by D.6)

Aim

To ensure that the system will meets its temporal and memory requirements.

Description

The requirements specification for the system and the software includes memory and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. An analysis is performed which will identify the distribution demands under average and worst case conditions. This analysis requires estimates of the resource usage and elapsed time of each system function. These estimates can be obtained in several ways, for example, comparison with an existing system or the prototyping and bench-marking of time critical systems.

B.53 Re-Try Fault Recovery Mechanisms (Referenced by clause 9)

Aim

To attempt functional recovery from a detected fault condition by re-try mechanisms.

Description

In the event of a detected fault or error condition, attempts are made to recover the situation by re-executing the same code. Recovery by re-try can be as complete as a re-boot and a re-start procedure or a small re-scheduling and re-starting task, after a software time-out or a task watchdog action. Re-try techniques are commonly used in communication fault or error recovery and re-try conditions could be flagged from a communication protocol error (check sum etc.) or from a communication acknowledgement response time-out.

Reference:

The Theory and Practise of Reliable System Design. D.P. Siewiorek and R.S. Schwarz, Digital Press.

B.54 Safety Bag (Referenced by clause 9)

Aim

To protect against residual specification and implementation faults in software which adversely affect safety.

Description

A safety bag is an external monitor, implemented on an independent computer to a different specification. This safety bag is solely concerned with ensuring the main computer performs safe, not necessarily correct, actions. The safety bag continuously monitors the main computer. The safety bag prevents the system from entering an unsafe state. In addition if it detects that the main computer is entering a potentially hazardous state, the system has to be brought back to a safe state either by the safety bag or the main computer.

References:

Using AI Techniques to Improve Software Safety. Proc. IFAC SAFECOMP 88, Sarlat, France, Oct 1986, Pergamon Press 1986.

B.55 Sneak Circuit Analysis (Referenced by D.8)

Aim

To detect an unexpected path or logic flow within a system which, under certain conditions initiates an undesired function or inhibits a desired function.

Description

A sneak circuit path may consist of hardware, software, operator actions, or combinations of these elements. Sneak circuits are not the result of hardware failure but are latent conditions inadvertently designed into the system or coded into the software programs, which can cause it to malfunction under certain conditions.

Categories of sneak circuits are:

– Sneak Paths which cause current, energy, or logical sequence to flow along an unexpected path or in an unintended direction;

– Sneak Timing in which events occur in an unexpected or conflicting sequence;

– Sneak Indications which cause an ambiguous or false display of system operating conditions, and thus may result in an undesired action by the operator;

– Sneak Labels which incorrectly or imprecisely label system functions, for example, system inputs, controls, displays, buses, etc., and thus may mislead an operator into applying an incorrect stimulus to the system.

Sneak circuit analysis, relies on the recognition of basic topological patterns with the hardware or software structure (e.g. six basic patterns are identified for software). Analysis takes place with the aid of a checklist of questions about the use and relationships between the basic topological components.

References:

Sneak Analysis and Software Sneak Analysis. S.G. Godoy and G.J. Engels, J. Aircraft Vol. 15, No. 8, 1978.

Sneak Circuit Analysis. J.P. Rankin, Nuclear Safety, Vol. 14, No. 5, 1973.

B.56 Software Configuration Management (Referenced by clause 15)

Aim

Software Configuration management aims to ensure the consistency of groups of development deliverables as those deliverables change. Configuration Management, in general, applies to both hardware and software development.

Description

Software Configuration Management is a technique used throughout development. In essence, it requires the recording of the production of every version of every "significant" deliverable and of every relationship between different versions of the different deliverables. The resulting records allow the developer to determine the effect on other deliverables of a change to one deliverable (especially on of its components). In particular, systems or subsystems can be reliably re-built from consistent sets of component versions.

References:

Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs. MIL-STD-483.

Software Configuration Management. J K Buckle, Macmillan Press, 1982.

Software Configuration Management. W A Babich, Addison-Wesley, 1986.

Configuration Management Requirements for Defence Equipment. UK Ministry of Defence Standard 05-57 Issue 1, 1980.

B.57 Strongly Typed Programming Languages (Referenced by clause 10)

Aim

Reduce the probability of faults by using a language which permits a high level of checking by the compiler.

Description

Such languages usually allow user-defined data types to be defined from the basic language data types (such as INTEGER, REAL). These types can then be used in exactly the same way as the basic types, but strict checks are imposed to ensure the correct type is used. These checks are imposed over the whole program, even if this is built from separately compiled units. The checks also ensure that the number and the type of procedure arguments match even when referenced from separately compiled modules.

Strongly typed languages also support other aspects of good software engineering practice such as easily analysable control structures (e.g. IF ..THEN .. ELSE, DO .. WHILE, etc) which lead to well-structured programs.

Typical examples of strongly typed languages are Pascal, Ada and Modula 2.

References:

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-815A 1983.

In search of Effective Diversity: a Six Language Study of Fault-Tolerant Flight Control Software, A Avizienis, M R Lyu, W Schutz, The Eighteenth International Symposium on Fault Tolerant Computing, Tokyo, Japan 27-30 June 1988, IEEE Computer Society Press,ISBN 0-8186-0867-6.

B.58 Structure Based Testing (Referenced by D.2)

Aim

To apply tests which exercise certain subsets of the program structure.

Description

Based on an analysis of the program a set of input data is chosen such that a large fraction of selected program elements are exercised. The program elements exercised can vary depending upon the level of rigour required.

- Statements: this is the least rigorous test since it is possible to execute all code statements without exercising both branches of a conditional statement.
- Branches: both sides of every branch should be checked. This may be impractical for some types of defensive code.
- Compound Conditions: every condition in a compound conditional branch (i.e. linked by AND/OR is exercised).
- LCSAJ: a linear code sequence and jump is any linear sequence of code statements including conditional jumps terminated by a jump. Many potential sub-paths will be infeasible due to constraints on the input data imposed by the execution of earlier code.
- Data Flow: the execution paths are selected on the basis of data usage for example a path where the same variable is both written and wrote.
- Call Graph: a program is composed of subroutines which may be invoked from other subroutines. The call graph is the tree of subroutine invocations in the program. Tests are designed to cover all invocations in the tree.
- Entire Path: execute all possible path through the code. Complete testing is normally infeasible due to the vary large number of potential paths.

References:

Reliability of the Path Analysis Testing Strategy. W. Howden, IEEE Trans Software Engineering, Vol SE 3, 1976.

B.59 Structure Diagrams (Referenced by D.5)

Aim

To show the structure of a program diagrammatically.

Description

Structure Diagrams are a notation which complements Data Flow Diagrams. They describe the programming system and a hierarchy of parts and display this graphically, as a tree. They document how elements of a data flow diagram can be implemented as a hierarchy of program units.

A structure chart shows relationships between program units without including any information about the order of activation of these units. They are drawn using the following three symbols:

i) a rectangle annotated with the name of the unit;

ii) an arrow connecting these rectangles;

iii) a circled arrow, annotated with the name of data passed to and from elements in the structure chart. Normally, the circled arrow is drawn parallel to the arrow connecting the rectangles in the chart.

From any non trivial data flow diagram, it is possible to derive a number of different structure charts.

Structure charts derived from data flow diagrams represent a first level structure of the system, where each box on the structure chart represents a bubble in the data flow diagram. Naturally, deeper levels can be described using the same technique.

References:

Software Engineering. I. Sommerville, Addison-Wesley 1982. ISBN 0-201-13795-X.

Structured Design. L.L Constantine and E, Yourdon, Englewood Cliffs, New Jersey, Prentice Hall, 1979.

Reliable Software Through Composite Design. G.J Myers, New York, Van Nostrand, 1975.

B.60 Structured Methodology (Referenced by clause 8 and clause 10)

Aim

The main aim of Structured Methodologies is to promote the quality of software development by focusing attention on the early parts of the life-cycle. The methods aim to achieve this through both precise and intuitive procedures and notations (assisted by computers) to identify the existence of requirements and implementation features in a logical order and a structured manner.

Description

A range of Structured Methodologies exist. Some such as SSADM, LBMS are designed for traditional data-processing and transaction processing functions, while others (MASCOT, JSD, real-time Yourdon) are more oriented to process-control and real-time applications (which tend to be more safety-critical).

Structured Methods are essentially "thought tools" for systematically perceiving and partitioning a problem or system. Their main features are:

– a logical order of thought, breaking a large problem into manageable stages;

– identification of total system, including the environment as well as the required system;

– decomposition of data and function in the required system;

– checklists, i.e. lists of the sort of things that need definition;

– low intellectual overhead - simple, intuitive, pragmatic.

The supporting notations tend to be precise for identifying problem and system entities (e.g. processes and data flows), but the processing functions performed by these entities tend to be expressed using informal notations. However some methods do make partial use of(mathematically) formal notations (for example JSD makes use of regular expressions: Yourdon, SOM and SDL utilise finite state machines). This precision not only reduces the scope for misunderstanding, it provides scope for automatic processing.

Another benefit of structured notation is their visibility, enabling a specification or design to be checked intuitively by a user, against his powerful but unstated knowledge.

This bibliography describes some of these Structured Methodologies in more detail, for instance, Controlled Requirements Expression, Jackson System Development, MASCOT, real-time Yourdon and Structured Analysis and Design Technique (SADT).

References:

Structured Development for real-time Systems (3 Volumes) P T Yourdon Press, 1985.

Essential Systems Analysis. St M McMenamin, F Palmer, Yourdon Inc, 1984. N Y.

The Use of Structured Methods in the Development of Large Software-Based Avionic Systems. D J Hatley, Proceedings DASC, Baltimore, 1984.

B.60.1 Controlled Requirements Expression (CORE)

Aim

To ensure that all the requirements are identified and expressed.

Description

This approach is intended to bridge the gap between the customer/end user and the analyst. It is not mathematically rigorous but aids the communication process - CORE is designed for requirements expression rather than specification. The approach is structured and the expression goes through various levels of refinement. The CORE process encourages a wider view of the problem, bringing in a knowledge of the environment in which the system will be used and the differing viewpoints of the various types of user. CORE includes guidelines and tactics for recognising departures from the 'grand design'. Departures can be corrected or explicitly identified and documented. Thus specifications may not be complete, but unresolved problems and high-risk areas are identified and have to be addressed in the subsequent design.

B.60.2 JSD - Jackson System Development

Aim

A development method covering the development of software systems from requirements through to code, with special emphasis on real-time systems.

Description

JSD is a staged development process in which the developer models the real world processes upon which the system functions are to be based, determines the required functions and inserts them into the model, and transforms the resulting specification into one that is realisable in the target environment. It therefore covers the traditional phases of definition, design and implementation but takes a somewhat different view from the traditional methods in not being top-down.

Moreover it places great emphasis on the early stage of identifying the entities in the real world that are the concern of the system being built and on modelling them and what can happen to them. Once this analysis of the 'real-world' has been done and a model created, the system's required functions are analysed to determine how they can fit into this real-world model. The resulting system model is augmented with structured descriptions of all the processes in the model and the whole is then transformed into programs that will operate in the target software and hardware environment.

References:

An Overview of JSD. J R Cameron, IEEE Trans SE-12 no 2 Feb 1986.

System Development. M Jackson, Prentice-Hall, 1983.

B.60.3 MASCOT

Aim

The design and implementation of real-time systems.

Description

MASCOT (Modular Approach to Software Construction, Operation and Test) is a design method supported by a programming system. It is a systematic method of expressing the structure of real-time system in a way that is independent of the target hardware or implementation language. It imposes a disciplined approach to design that yields a highly modular structure, ensuring a close correspondence between the functional elements in the design and the construction elements appearing in system integration. A system is designed in terms of a network of concurrent processes that communicate through channels. Channels can be either pools of fixed data or queues (pipelines of data). Control of access to channels is defined independently of the processes. It is defined in terms of access mechanisms that also enforce scheduling rules on the processes. Recent versions of MASCOT have been designed with Ada implementation in mind.

MASCOT supports an acceptance strategy based on the test and verification of single modules and larger collections of functionally related modules. A MASCOT implementation is intended to be built upon a MASCOT kernel - a set of scheduling primitives that underline the implementation and support the access mechanisms.

Reference:

MASCOT 3 User Guide. MASCOT Users Forum, RSRE, Malvern, England, 1987.

B.60.4 Real-time Yourdon

Aim

This aims to be a complete software development method consisting of specification and design techniques oriented towards the development of real-time systems.

Description

The development scheme underlying this technique assumes a three phase evolution of a system being developed. The first phase involves the building of an 'essential model', one that describes the behaviour required by the system. The second involves the building of an implementation model which describes the structures and mechanisms that, when implemented, embody the required behaviour. The third phase involves the actual building of the system in hardware and software. The three phases correspond roughly to the traditional definition, design and implementation phases but lay greater emphasis on the fact that at each stage the developer is engaged in a modelling activity.

The essential model is in two parts: the environmental model containing a definition of the boundary between the system and its environment and a description of the external events to which the system must respond, and the behavioural model which contains schemas defining the transformation the system carries out in response to events and a definition of the data the system must hold in order to respond.

The implementation model also divides into sub-models: in this case covering the allocation of individual processes to processors and of the decomposition of the processes into modules.

To capture these models the technique combines a number of well-known techniques: data-flow diagrams, structured English, state transition diagrams and Petri Nets.

Additionally the method contains techniques for simulating a proposed system design either on paper or mechanically from the models that are drawn up.

Reference:

Structured Development for Real-time Systems (3 Volumes) PT Ward and SJ Mellor. Yourdon Press, 1985.

B.60.5 SADT- Structured Analysis and Design Technique

Aim

To model and identify, in a diagrammatic form using information flows, the decision making processes and the management tasks associated with a complex system.

Description

In SADT, the concept of an Activity-Factor Diagram plays a central role. An A/F diagram consists of activities grouped in so called 'action boxes'. Each action box has a unique name, and is linked to other action boxes by factor relations (drawn as arrows) which are also given unique names. Each action box can be hierarchically decomposed into subsidiary action boxes and relations. There are four types of factors: inputs, controls mechanisms and outputs:

– Input: indicated by an arrow that enters an action box at the left hand side. Inputs can represent material or immaterial things and they are suitable for manipulation by one or more activities in an action box;

– Control: are typically instructions, procedures, choice criteria and so on. Controls guide the execution of an activity and they are shown by arrows entering the top side of an action box;

– Mechanism: is a resource such as personnel, organisational units or equipment, that is needed for an activity to perform its task;

– Output: can denote anything that an activity produces, and it is pictured by an arrow leaving an action box at the right hand side.

When activities are strongly related to each other by many factor relations then it perhaps better to consider these activities as an indivisible group that is contained in one action box which does not lend itself to further detailing of its content. The guiding principle for grouping of activities into action boxes is that the resulting boxes are coupled pairwise by only a few factors.

The model hierarchy of A/F diagrams is pursued until a further detailing of the action boxes is meaningless. This stage is reached when the activities within the boxes are inseparable or when further detailing of the action boxes falls outside the scope of the system analysis.

References:

Structured Analysis for Requirements Definition, DT Ross, KE Schoman Jr, IEEE Trans.Software Eng. Vol SE-3, 1977, 6-15.

Structured Analysis (SA): A language for communicating ideas, DT Ross, IEEE Trans.Software Eng., Vol SE-3,1, 16-34.

Applications and Extensions of SADT, DT Ross, Computer, April 1985, 25-34.

Structured Analysis and Design Technique - Application on Safety Systems, W Heins, Risk Assessment and Control Courseware, Module B1, chapter 11. 1989, Delft University of Technology, Safety Science Group, PO Box 5050, 2600 GB Delft, Netherlands.

B.61 Structured Programming (Referenced by clause 10)

Aim

To design and implement the program in a way which makes practical the analysis of the program. This analysis should be capable of discovering all significant program behaviour.

Description

The program should contain the minimum of structural complexity. Complicated branching should be avoided. Loop constraints and branching should (where possible) be simply related to input parameters. The program should be divided into appropriately small modules, and the interaction of these modules should be explicit. Features of the programming language which encourage the above approach should be used in preference to other features which are (allegedly) more efficient, except where efficiency takes absolute priority (e.g. some safety-critical systems).

References:

A Discipline of Programming. E W Dijkstra, Englewood Cliffs N J, Prentice-Hall, 1976.

Assessing a Class of Software Tools. M A Hennell et al, 7th International conference on Software Engineering, March 1984, Orlando.

A Software Tool for Top-down Programming. D C Ince, Software - Practice and Experience, Vol 13, No 8, August 1983.

B.62 Suitable Programming Languages (Referenced by D.4)

Aim

To support the requirements of this International Standard as much as possible, in particular, defensive programming, strong typing, structured programming and possibly assertions. The programming language chosen should lead to easily verifiable code with a minimum of effort and facilitate program development, verification and maintenance.

Description

The language should be fully and unambiguously defined. The language should be user or problem oriented rather than machine oriented. Widely used languages or their subsets are preferred to special purpose languages.

In addition to the already referenced features the language should provide for

- block structure,
- translation time checking,
- run time type and array bound checking, and
- parameter checking.

The language should encourage

- the use of small and manageable modules,
- restriction of access to data in defined modules,
- definition of variable sub-ranges, and
- any other type of error limiting constructs.

It is desirable that the language is supported by a suitable translator, appropriate libraries of pre-existing modules, a debugger and tools for both version control and development.

Features which make verification difficult and therefore should be avoided are:

- unconditional jumps excluding subroutine calls;
- recursion;
- pointers, heaps or any type of dynamic variables or objects;
- interrupt handling at source code level;
- multiple entries or exits of loops, blocks or subprograms;
- implicit variable initialisation or declaration;
- variant records and equivalence; and
- procedural parameters.

Low level languages, in particular assembly languages, present problems due to their machine oriented nature.

B.63 Symbolic Execution (Referenced by D.8)

Aim

To show the agreement between the source code and the specification.

Description

The program is executed substituting the left hand side by the right hand side in all assignments. Conditional branches and loops are translated into Boolean expressions. The final result is a symbolic expression for each program variable. This can be checked against the expected expression.

References:

Formal Program Verification using Symbolic Execution. R.B. Dannenberg and G.W. Ernst,IEEE Trans Software Engineering, Vol. SE-8, No 1, 1982.

Symbolic Execution and Software Testing. J.C. King, Comm. ACM, Vol. 19, No 7, 1976.

B.64 Time Petri Nets (Referenced by D.5 and D.7)

Aim

To model relevant aspects of the system behaviour and to assess and possibly improve safety and operational requirements through analysis and re-design.

Description

Petri nets belong to a class of graph theoretic models which are suitable for representing information and control flow in systems exhibiting concurrency and asynchronous behaviour.

A Petri net is a network of places and transitions. The places may be 'marked' or 'unmarked'. A transition is 'enabled' when all the input places to it are marked. When enabled, it is permitted (but not obliged) to 'fire'. If it fires, the input marks are removed, and each output place from the transition is marked instead.

The potential hazards are represented as particular states (markings) in the model. Extended Petri nets allow timing features of the system to be modelled. Although 'classical' Petri nets concentrate on control flow aspects, several extensions have been proposed to incorporate dataflow into the model.

References:

Net Theory and Applications. W. Brauer (ed), Lecture Notes in Computer Science, Vol. 84, Springer 1980.

Petri Net Theory and Modelling of Systems. J.L. Peterson, Prentice Hall, 1981.

Safety Analysis using Petri Nets. N. Leveson and J. Stolzy, Proc. FTCS 15, Ann Arbor,Michigan, June 1985, IEEE 1985.

A Tool for Requirements Specification and Analysis of Real Time Software Based on Timed Petri Nets. S. Bologna, F. Pisacane, C Ghezzi, D. Mandrioli, Proc. SAFECOMP 88, 9-11Nov. 1988. Fulda Fed. Rep. of Germany 1988.

B.65 Translator Proven In Use (Referenced by clause 10)

Aim

To avoid any difficulties due to translator failures which can arise during development, verification and maintenance of a software package.

Description

A translator is used, whose correct performance has been demonstrated in many projects already. Translators without operating experience or with any serious known errors are prohibited.

If the translator has shown small deficiencies the related language constructs are noted down and carefully avoided during a safety related project.

Another version to this way of working is to restrict the usage of the language to only its commonly used features.

This recommendation is based on the experience from many projects. It has been shown that immature translators are a serious handicap to any software development. They make a safety-related software development generally infeasible.

It is also known, presently, that no method exits to prove the correctness for all compiler parts.

B.66 Walkthroughs / Design Reviews (Referenced by D.8)

Aim

To detect errors in some product of the development process as soon and as economically as possible.

Description

IEC/TC 56, have published a Guide on Formal Design Reviews, which includes a general description of formal design reviews, their objectives, details of the various design review types, the composition of a design review team and their associated duties and responsibilities. The IEC document also provides general guidelines for planning and conducting formal design reviews, as well as specific details concerning the role of independent specialists within a design review team. Examples of specialist functions include, amongst others, Reliability, Maintenance Support and Availability.

The IEC recommend that a "formal design review shall be conducted for all new products/processes, new applications, and revisions to existing products and manufacturing processes which affect the function, performance, safety, reliability, ability to inspect maintainability, availability, ability to cost, and other characteristics affecting the end product/process, users or bystanders".

A code walk through consists of a walk through team selecting a small set of paper test cases, representative sets of inputs and corresponding expected outputs for the program. The test data is then manually traced through the logic of the program.

References:

The Art of Software Testing. G. Myers, Wiley & Sons, New York, 1979.

Reliability and Maintainability Guide on Formal Design Review (Draft) International Electrotechnical Commission, Technical Committee No 56, January 1988.

B.67 Fuzzy Logic (Referenced by clause 10)

Aim

Fuzzy Logic is a mathematical discipline, based on the fuzzy set theory that allows for degrees of truth and falseness. It is a generalisation of bi-level logic which provides a model for approximate reasoning. It enables the incorporation of human intelligence into automatic systems. By acknowledging the difficulty of defining precise boundaries, Fuzzy Logic reduces the required precision, and hence leads to high level and simple solutions which are easy to control.

Description

The essential part of a Fuzzy Logic solution is a set of linguistic rules (IF - THEN rules) where the antecedents and the consequents are associated with fuzzy sets.

EXAMPLE IF speed is fast and distance_to_stop is medium
THEN accelerator is near zero and brake is light

In this example, "speed" is a linguistic variable characterised by a fuzzy set "fast", which can be interpreted as "speed is above about 70 km/h". If speed is less than 60 km/h, the membership value of "fast" is 0. If speed is above 80 km/h, the memberships value of "fast" is 1. If speed is between 60 and 80 km/h, the membership value of "fast" varies between 0 and 1.

The decision making logic is based on mathematical classes of operators: the triangular norms and triangular co-norms.

Fuzzy Logic rule-based systems differ from other expert systems in many ways as: (1) the small number of rules, (2) the use of forward chaining only, (3) non-chaining of inferences (all rules are executed in parallel in one iteration), (4) the statistically defined rules, (5) the speed of execution due to simplicity of the solutions, and (6) the determinism.

During the past few years, Fuzzy Logic has found numerous applications in fields ranging from finance to earthquake engineering. In particular, fuzzy control has emerged to control highly non-linear systems, or systems which mathematical description is unknown or too complex to be treated analytically, or systems in which the available measurements are of poor quality.

Notable applications of fuzzy logic control include aircraft flight control, and power systems and nuclear reactor control. More recently, fuzzy control has been applied successfully to automatic train operation systems.

References:

Fuzzy Sets: Zadeh. Information and Control, 1965, vol 8, pp338-353

Fuzzy Logic in Control Systems : Fuzzy Logic Controller, Part I & II. Chuen Chien Lee. IEEE Transactions on systems, man, and cybernetics, vol. 20, No2, March/April 1990

Industrial Applications of Fuzzy Control : M.Sugeno Ed., Amsterdam North Holland, 1985

Automatic Train Operation System by Predictive Fuzzy Control : Yasunobu, Miyamoto, in Industrial Applications of Fuzzy Control, M.Sugeno Ed., Amsterdam North Holland, 1985

An automatic operation method for control rods in BWR plants : Kinoshita, Fukusaki, Satoh, Miyake, Proc. Specialists Meeting on In-Core Instrumentation and Reactor Core Assessment. Cadarache, France 1988

Use of rule-based system for process control. Bernard. IEEE Contr. Syst. Mag., Vol.8, No.5, pp.3-13, 1988

B.68 Object Oriented Programming (Referenced by clause 10)

Aim

To enable rapid prototyping, to more easily reuse existing software components, to achieve information hiding, to reduce the likelihood of errors during the whole lifecycle, to reduce the necessary effort during the maintenance phase, to break down complex problems into more easily manageable small problems, to reduce the dependencies between software components, to create more easily extendible applications.

Description

Object oriented programming is a fundamentally new way of thinking about software based on abstractions that exist in the real world rather than based on computational abstractions. Object oriented programming organises software as a collection of objects that incorporate both data structure and behaviour. This is in contrast to conventional programming where data structure and behaviour are only loosely connected.

Object: an object consists of a private data area and set of operations - so called methods - on that object. Methods may be public or private. No other software component is allowed to read or change the private data of an object directly. Every other software component has to use the public methods on that object to read or write data in the private data area of an object.

Object Class: by specifying an object class (often in the form of a type definition) you enable the instantiation of numerous objects of the same class, i.e., all instantiations have the private data area and the methods defined in the object class.

(Multiple) Inheritance: an object class can inherit the private data area and the methods of one (or more) superclasses (object classes above it in the class hierarchy) with being allowed to add some private data, to add some methods or to modify the implementations of the inherited methods. Using Inheritance multiple object class trees can be built.

Polymorphism: the same operation may behave different on different object classes, e.g., the write operation for a terminal object writes characters to that terminal and a write operation to a file object writes characters to that file.

References:

Object Oriented Software Construction, Bertrand Meyer, England: Prentice Hall International, 1988.

Classification as a paradigm for computing, Peter Wegner, Technical Report CS-86-11, Brown University, May 1986.

Learning language, Peter Wegner, Byte (McGraw-Hill publication) March 1989.

All OOPSLA (Object-Oriented Programming Systems, Languages and Applications) and ECOOP (European Conference on Object-Oriented Programming) conference proceedings.

B.69 Traceability (Referenced by clause 11)

Aim

The objective of Traceability is to ensure that all requirements can be shown to have been properly met and that no untraceable material has been introduced.

Description

Traceability to requirements shall be an important consideration in the validation of a system and means shall be provided to allow this to be demonstrated throughout all phases of the lifecycle.

Traceability shall be considered applicable to both functional and non-functional requirements and shall particularly address

a) traceability of requirements to the design or other objects which fulfil them,

b) traceability of design objects to the implementation objects which instantiate them,

c) traceability of requirements and design objects to the operational and maintenance objects required to be applied in the safe and proper use of the system,

d) traceability of requirements, design, implementation, operation and maintenance objects, to the verification and test plans and specifications which will determine their acceptability,

e) traceability of verification and test plans and specifications to the test or other reports which record the results of their application.

Where requirements, design or other objects are instantiated as a number of separate documents, traceability shall be maintained within the document structures and in a hierarchical manner.

The output of the Traceability process shall be the subject of formal Configuration Management.